Interpreter
&
Buffer Manager
&
MINISQL Schema
设计总报告

邓墨琳 3150103457

计算机科学与技术 1502

时间：2018-06

目录

第 1 章　　MINISQL 总体框架

第 1.1 节　MiniSQL　实现功能分析

我们设计并实现了一个精简型单用户 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；索引的建立/删除以及表记录的插入/删除/查找。

1．数据类型

只要求支持三种基本数据类型：int，char(n)，float，其中 char(n)满足 1 <= n <= 255 。

2．表定义

一个表最多可以定义 32 个属性，各属性可以指定是否为 unique；支持单属性的主键定义。

3．索引的建立和删除

对于表的主属性自动建立 B+树索引，对于声明为 unique 的属性可以通过 SQL 语句由用户指定建立/删除 B+树索引（因此，所有的 B+树索引都是单属性单值的）。

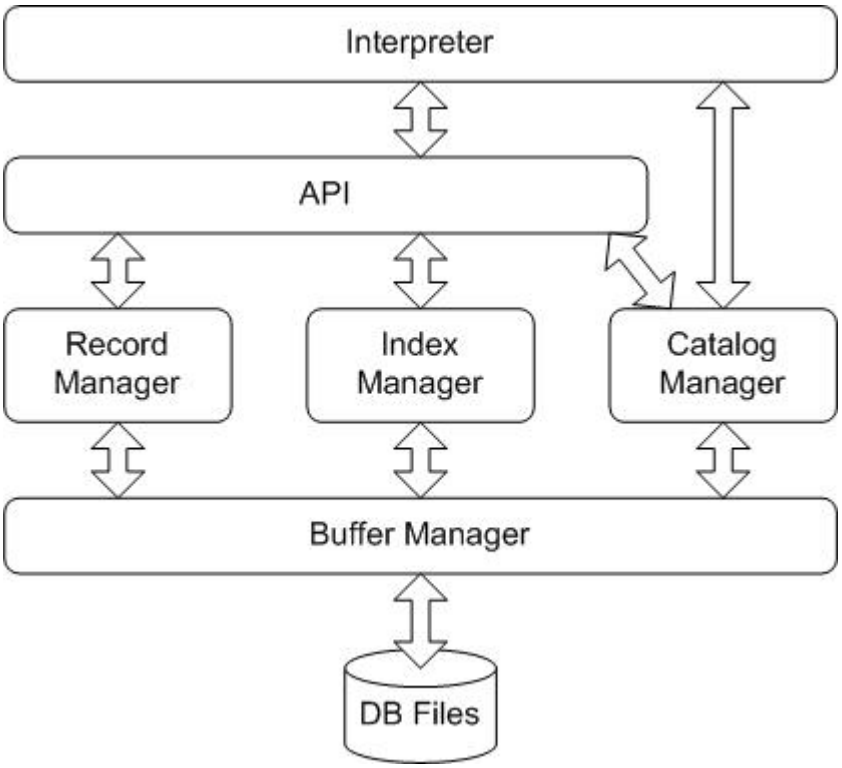
4．查找记录

可以通过指定用 and 连接的多个条件进行查询，支持等值查询和区间查询。

5．插入和删除记录

支持每次一条记录的插入操作；支持每次一条或多条记录的删除操作。

第 1.2 节 MiniSQL 系统体系结构



MiniSQL 体系结构

第 1.3 节 设计语言与运行环境

工具： C++

环境： g++ 4.9.2 or higher
bison 3.0.4
flex 2.6.4

第 2 章 数据结构

2.1 节 数据库 schema

1.字段属性：

```
struct Attribute {
    string attr_name;
    int attr_type;
    int attr_key_type;
    int attr_len;
    int attr_id;
};
```

`attr_type:int` 字段数据类型，分别为 CHAR,FLOAT,INT

`attr_key_tpe:int` 完整性约束信息，分别为 PRIMARY,UNIQUE,NULL

`attr_len:int` 字段长度，CHAR 为 1，其余为 4

`attr_id:int` 字段 id，记录位于表中位置

2.Table 描述 relation schema：

```
struct Table {
    string table_name;
    int attr_count;
    Attribute attrs[32];
    ...//some functions
}
```

`table_name:string` 表名

`attr_count:int` 表中字段个数

`attrs:Attribute[32]` 表中字段列表，最多 32 个属性

3.查询条件信息：

```
struct Condition {
    string attr_name;
    int attr_type;
    string op_type;
    string cmp_value;
};
```

`attr_name:string` 字段名

`attr_type:int` 字段对应的数据类型

`op_type:string` 运算符类型，分别为<>,=,<=,<,>,>=

`cmp_value:string` 操作数，进行比较的值

4.查询条件信息表：

```
typedef list<Condition> Condition_list;
```

包含多条查询信息。

5.索引信息

```
struct Index {
    string index_name;
    string table_name;
    string attr_name;
};
```

`index_name:string` 索引名

`table_name:string` 表名

`attr_name:string` 索引对应的字段名

6.索引名

```
struct IndexName {
    string name[100];
    int len;
};
```

单独储存索引名

7.记录

```
struct Record {
    string table_name;
    vector<string> attr_values;
    int num_values;
};
```

`table_name:string` 表名

`attr_values:vector<string>` 一条记录的值

`num_values:int` 插入记录的字段个数，应与 table 中字段数相匹配

2.2 节 Interpreter：

SQL 语句抽象语法树(实际为 concrete syntax tree)

```
struct CST {
    int type;
    char text[256];
    CST *lpNext;
    CST *lpSub;
};
```

`type:int` 抽象语义动作，对应 SQL 查询语句类型，分别为 CREATE,DROP,INSERT,DELETE,SELECT,EXECFILE(创建删除包括表和 index)

`text:char[256]` 语义值

`lpNext: CST*` 指向兄弟节点

`lpSub: CST*` 指向子节点

2.3 节 Buffer Block 数据结构：

数据块 Block：

```
struct Block {
    string file_name;
    int id;
    bool dirty;
    bool pin;
    char data[MAX_BLOCK_SIZE];
    // C'tor
    Block(const char* _file_name, int _id):
        file_name(_file_name), id(_id) {
        dirty = pin = false;
    }
};
```

`id:int` block 的 index

`dirty:bool` 标记是否为脏页

`pin:bool` 标记是否该 block 被锁定

`data:char[]` block 中的数据，以二进制读入和存放

`Block():C'tor` 构造函数

数据块双向链表：

```
struct Block_node {
    Block* block;
    Block_node* pre;
    Block_node* next;
    // C'tor
    Block_node(Block* _block): block(_block) {}
    // D'tor
    ~Block_node() { remove_node(); }
    // Add to position after node_
    void attach(Block_node* node_) {
        pre = node_; next = node_->next;
        node_->next->pre = this; node_->next = this;
    }
    // Remove from linked list
    void remove_node() { pre->next = next; next->pre = pre;}
};
```

`block:Block*` 指向 block 的指针

`pre:Block_node*` 指向链表前一个结点

`next:Block_node*` 指向链表下一个结点

`Block_node():C’tor` 构造函数

`~Block_node():D’tor` 析构函数

`remove_node():void` 删除当前结点

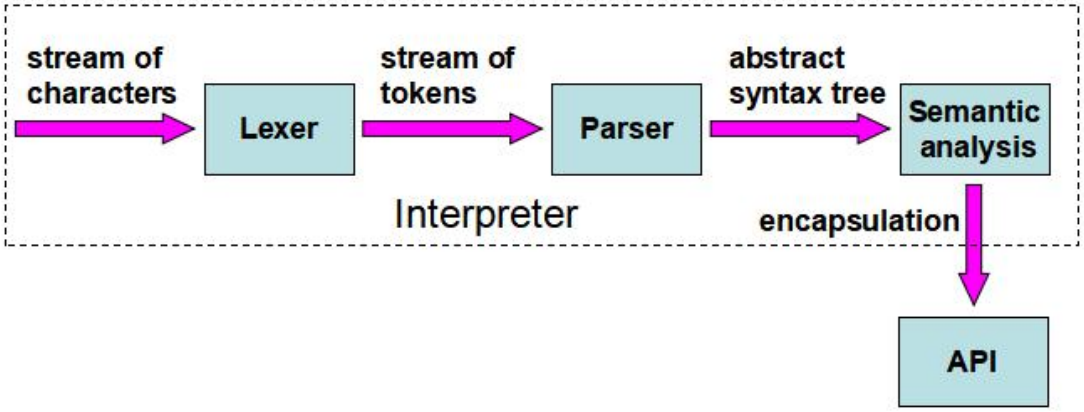
第 3 章　模块设计

3.1 节 Interpreter 设计

3.1.1 概述

本组 miniSQL 解释器通过 LEX、YACC 实现，LEX 进行词法分析，将字符流转化为 token 流。YACC 进行语法分析，构建抽象语法树，并通过语义分析得到语义值封装成 SQL schema 数据结构，传给 API 相关接口，来实现语义动作。通过在 Interpreter 类中调用 YYPARSE 函数可完成这一功能。

简易流程图如下所示



3.1.2 编译环境

Ubuntu 16.04LTS

g++　　4.9.2

bison　3.0.4

flex　　2.6.4

3.1.3 词法分析

(1) 关键字

CREATE、SELECT、WHERE、DROP、FROM、PRIMARY、AND、ON、TABLE、INDEX、PRIMARY、KEY、VALUES、INSERT、INTO、DELETE、QUIT、EXECFILE、INT、FLOAT、CHAR、NULL

(2) 正则表达式

以 float 为例，正则表达式为

```
-?[0-9]+"."[0-9]*                |
-?"."[0-9]+                      |
-?[0-9]+E[-+]?[0-9]+             |
-?[0-9]+"."[0-9]*E[-+]?[0-9]+    |
-?"."[0-9]+E[-+]?[0-9]+
```

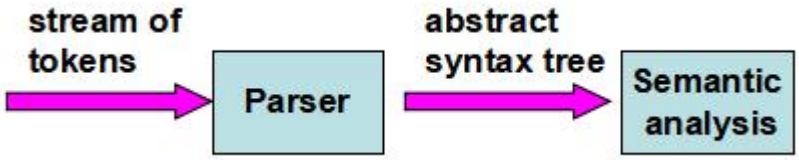
(3) 归约动作

LEX 读取字符流过程中，遇到匹配项，执行相应的归约动作。以 float 为例。

```
finished_state = 0;
yylval.pNode = MallocNewNode();
yylval.pNode->type=FLOATNUM;
snprintf(yylval.pNode->text,sizeof(yylval.pNode->text),"%s",yytext);
return FLOATNUM;
```

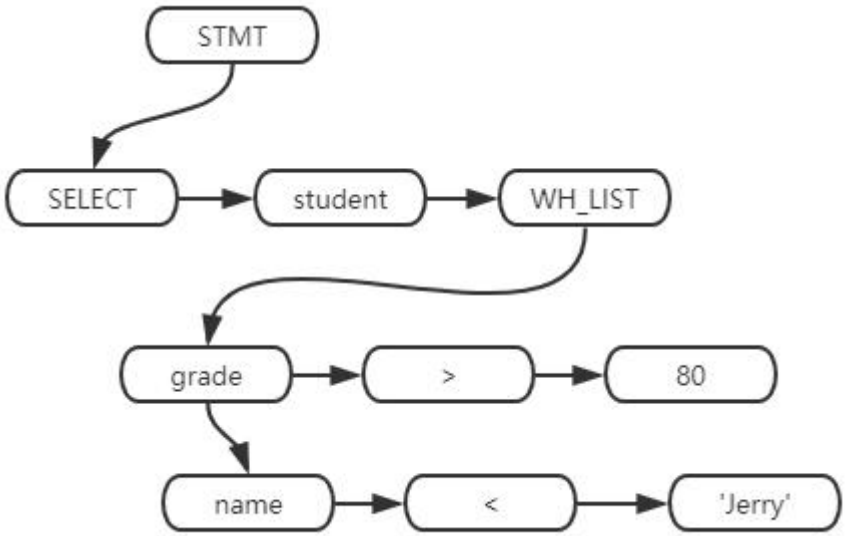
上述代码中，匹配到一个浮点数之后，创建了一个节点，将 float 值存入该节点，返回一个 token(YACC 根据文法生成对 token 进行相应的处理)

3.1.4 语法分析



miniSQL 的文法生成规则见附录。

以 `select * from student where grade > 80 and name < 'Jerry';` 为例展示抽象语法树

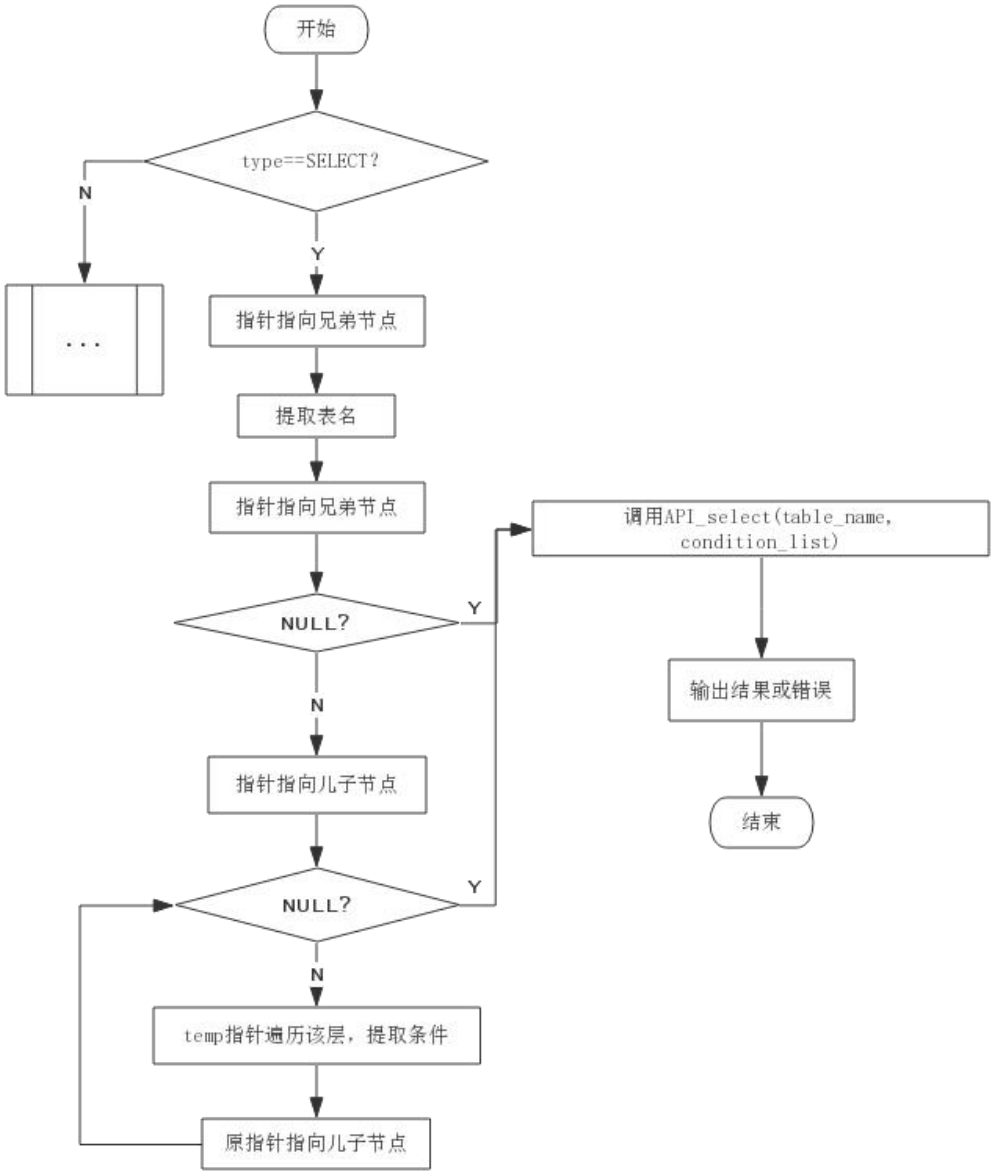


3.1.5 语义提取

YACC 自底向上到达“顶层”时，调用 ExecTree 函数解析树，提取语义信息。

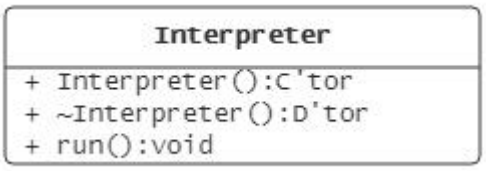
```
stmt_list: stmt ';' {ExecTree($1);nm_clear();} |
stmt_list stmt ';'  {ExecTree($2);nm_clear();};
```

同样以 `select * from student where grade > 80 and name < ‘Jerry’;` 为例，以下流程图展示了 CST.cpp 中的_Select 函数如何遍历抽象语法树，从中提取语义信息，封装成 schema 结构，最后调用 API 执行相关语义动作。



其他操作类似。

3.1.6 Interpreter 类



`run():void` 调用 yyparse()函数，完成解释器功能。

```
void run() {
   yyparse();
}
```

3.1.7 错误处理

由 yyerror()函数处理

```
void yyerror(const char *s, ...) {
    va_list ap;
    va_start(ap, s);
    fprintf(stderr, "line %d: error near '%s': ", yylineno, yytext);
    vfprintf(stderr, s, ap);
    fprintf(stderr, "\n");
    va_end(ap);
}
```

当 yywrap()函数检测到语法错误时，调用 yyerror 函数输出错误信息，同时停止解析，释放语法树空间。

yylineno、yytext 均由 YACC 提供，分别表示挡墙解析的行号和出错时的文本值。

```
stmt_list: error ';'        {yyerror("grammar error."); nm_clear();}
|   stmt_list error ';'     {yyerror("grammar error."); nm_clear();}
```

3.2 节 Buffer Manager 设计

3.2.1 功能描述

Buffer Manager 负责内存中 block 管理，实现功能有

1. 从 disk 中读取数据到内存中；将内存中的数据写入 disk

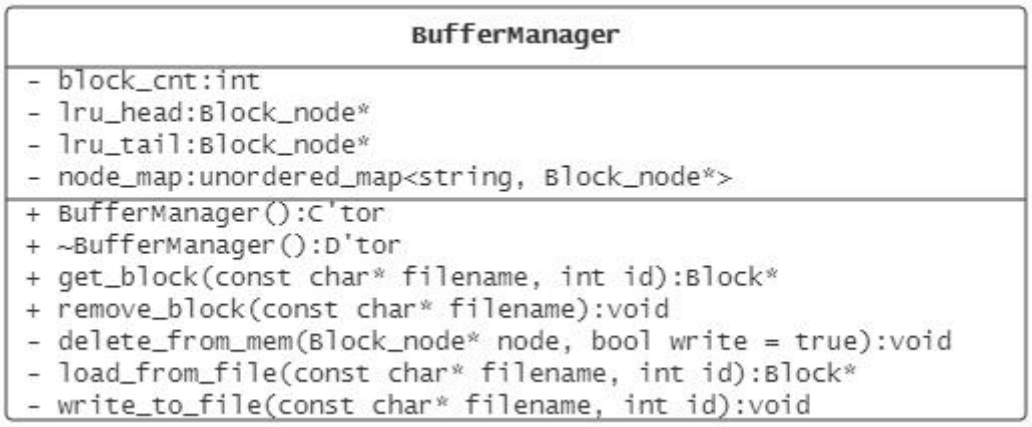
2. 实现 LRU 调度算法,当缓冲区满时选择 least recent used block swap out

3. 记录 block 是否为脏页

4. 提供 pin 功能

本项目中定义最大 block 大小为 4KB，最大 block 数(缓冲区)为 100.

3.2.2 BufferManager 类



`-block_cnt:int` 当前内存中 block 数量

`-lru_head:Block_node*` 双向链表伪头部

`-lru_tail:Block_node*` 双向链表伪尾部

`-node_map:unordered_map<string, Block_node*>` 索引信息

`+BufferManager():C'tor` 构造函数

`+~BufferManager():D'tor` 析构函数

`+get_block(const char* filename, int id):Block*` 获取内存中第 id 个 block

`+remove_block(const char* filename):void` 删除内存上所有以 filename 命名的 block

`-delete_from_mem(Block_node* node, bool write = true):void` 删除双向链表中该节点以及内存中的 block

`-load_from_file(const char* filename, int id):Block*` 从 disk 中读取第 id 个 block

`-write_to_file(const char* filename, int id):void` 将 block 写入 disk

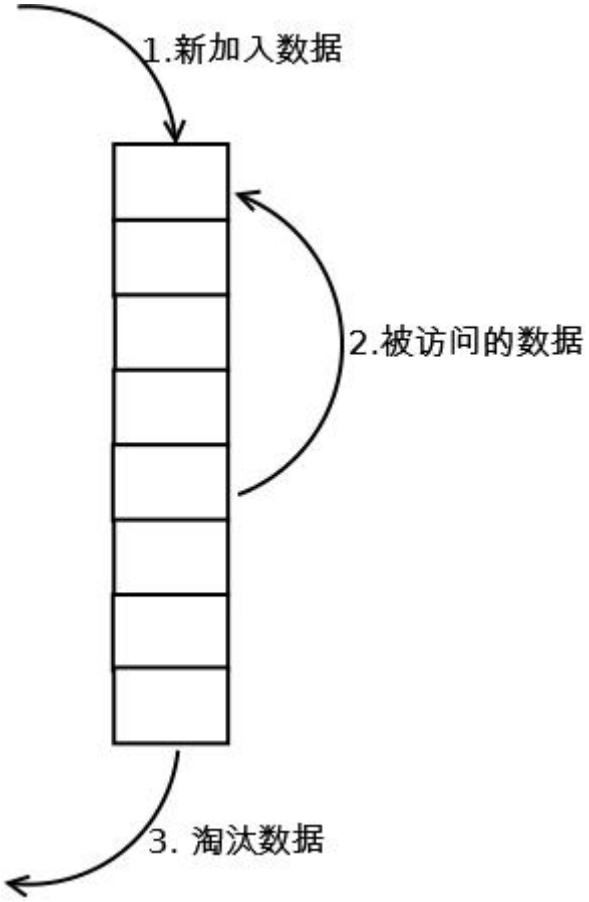
3.2.3 LRU 调度算法实现

数据结构为双向链表，头尾均添加 dummy pointers 方便管理。swap out 规则类似于 FIFO。

新加入的 block 插入链表头；

新被访问的数据插入链表头；

swap out 从尾部直接淘汰



在链表头插入 node 的方法由结构体 Block_node 内部实现。

```
void attach(Block_node* node_) {
    pre = node_; next = node_->next;
    node_->next->pre = this; node_->next = this;
}
```

3.2.4 Buffer Manager 接口

`Block* get_block(const char* filename, int id)`

获取内存中第 id 个 block，同时将该 block 置换至链表头；如果缓存区已满(链表长度达到最大长度)，从链表尾部将 block 删除，同时写入 disk。

```
Block* BufferManager::get_block(const char* filename, int id) {
    string block_name = string(filename) + "." + to_string(id);
    if (node_map.find(block_name) != node_map.end()) {
        // Set block as most recently used
        Block_node* node = node_map[block_name];
        node->remove_node();
        node->attach(lru_head);
        return node->block;
    }
      if (block_cnt == MAX_BLOCK_COUNT) {
    // Current block number full
    // Find the least recently used block
        Block_node* node = lru_tail->pre;
        while (node->block->pin)
        node = node->pre;
        delete_from_mem(node);
    }
    return load_from_file(filename, id);
}
```

`void remove_block(const char* filename)`

在内存中删除以 filename 命名的 block，同时将它们写入 disk

```
void BufferManager::remove_block(const char* filename) {
    Block_node* next_node;
    for (Block_node* node = lru_head->next; node != lru_tail; node =
next_node)
    {
        next_node = node->next;
        if (node->block->file_name == filename)
            delete_from_mem(node, false);
    }
}
```

第 4 章 附录

miniSQL 文法生成规则

```
stmt_list: error ';'
|   stmt_list error ';'

stmt_list: stmt ';'
|   stmt_list stmt ';'
;

stmt: CREATE TABLE NAME '(' attr_info ')'
|   DROP TABLE NAME
|   CREATE INDEX NAME ON NAME '(' NAME ')'
|   INSERT INTO NAME VALUES '(' attr_value_list ')'
|   DELETE FROM NAME
|   DELETE FROM NAME WHERE wh_list
|   SELECT '*' FROM NAME
|   SELECT '*' FROM NAME WHERE wh_list
|   EXECFILE NAME
|   QUIT
;

attr_info: attr_list
|   attr_list ',' PRIMARY KEY '(' NAME ')'
;

attr_list: attr
|   attr_list ',' attr
;

attr: NAME data_type
|   NAME data_type UNIQUE
;

data_type: INT
|   FLOAT
|   CHAR '(' INTNUM ')'
;

attr_value_list: real_value
|   attr_value_list ',' real_value
;

wh_list: wh_name
```

```
|   wh_list AND wh_name
wh_name: NAME COMPARISON real_value
real_value: STRING
|   INTNUM
|   FLOATNUM
|   EMPTY
;
```

第 5 章 References

[1]Modern Compiler Implementation in C

[2]LRU 算法

[3]Database System Concepts