具体设计报告

海杰文 3130000656

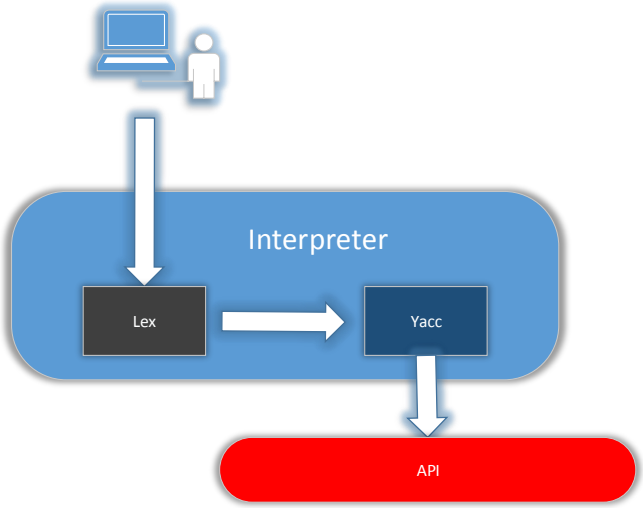
- # Interpreter

1. 设计原则

Interpreter 作为与用户交流的界面，需要拥有正确、有效的命令解析能力，并且提供一个友好的命令提示回显以及报错功能。为了程序的健壮性、可扩展性以及开发的效率，我使用 lex 和 yacc 这两个语法分析以及词法分析器。通过这两个工具，我构建了良好的用户交互界面。

2. 具体设计

整个 Interpreter 被一个上层的对象封装起来，通过进一步的封装，Lex 和 Yacc 可以很好地被嵌入到我们系统的体系中来。



前端的所有词语都在 Lex 中进行分析，而后被传到 Yacc 进行语法分析，Yacc 的语法分析成功后将调用相应的 API 进行处理。

这就是整个 Interpreter 的基本操作流程，程序的入口仅需保证 Interpreter 的正确执行就可以确保所有的 SQL 指令都被正确执行。

3. 代码实现

Interpreter 的正确运转依赖于两个方面，第一个是词法的正确分析以及转换，另一个是语义的正确分析，以便 SQL 指令的解释执行。

Token:

下面是所有 WickyDB 所需要的固定 token，这些 token 的解析构成了整个语义解析的基础。

```
select   { return token::SELECT; }
insert   { return token::INSERT; }
delete   { return token::DELETE; }
create   { return token::CREATE; }
drop     { return token::DROP; }
table    { return token::TABLE; }
```

```
index    { return token::INDEX; }
values   { return token::VALUES; }
null     { return token::NULLX; }
COMPARISON { return token::COMPARISON; }
from     { return token::FROM; }
where    { return token::WHERE; }
or               { return token::OR; }
and              { return token::AND; }
not      { return token::NOT; }
primary{ return token::PRIMARY; }
key      { return token::KEY; }
all              { return token::ALL; }
distinct {return token::DISTINCT; }
on               { return token::ON; }
unique   { return token::UNIQUE; }
into     { return token::INTO; }
int              { return token::INT; }
char     { return token::CHAR; }
float    { return token::FLOAT; }
exit     { return token::EXIT; }
execfile { return token::EXEC; }
show     { return token::SHOW; }
desc     { return token::DESC; }
```

类型解析：

```
        /* numbers */
[0-9]+  {
        long n = strtol (yytext, NULL, 10);
        if (! (INT_MIN <= n && n <= INT_MAX && errno != ERANGE))
                driver.error (*yylloc, "integer is out of range");
        yylval->strval = new std::string (yytext);
```

```
        return token::INTNUM;
};
```

上面这段代码可以很好地解析整数，当然，处于后端 API 调用的统一，我们将所有的类型都存成字符串。同样是因为为了后端的方便，我们分别解析了这些不同类型的数据以得到标记而后端不需要自行判断。

```
[0-9]+"."[0-9]* |
"."[0-9]*           {
        double n = strtod (yytext, NULL);
        if (errno == ERANGE)
                driver.error (*yylloc, "float is out of range");
        yylval->strval = new std::string (yytext);
        return token::APPROXNUM;
};
```

这段代码用来匹配小数并进行解析。

```
        /* strings */
'[^'\n]*'{
        yylval->strval = new std::string (yytext);
        return token::STRING;
};
```

这段代码用来匹配字符串并进行解析。

SQL 语法分析：

用于代码过长，所以我仅列举较为复杂的 select 语法分析。下面这段代码同样包括 API 层一些调用，可以看出前段的 Interpreter 是通过怎样的方法与后端相连，将指令转化为 API 的调用最终执行整个程序。

```
select_statement:
                SELECT opt_all_distinct selection table_exp {
                        {
                                Table* t1 = driver.table;
                                WickyEngine* we = WickyEngine::getInstance();
                                try {
                                        driver.table = we->Select(t1, *(driver.getCondition()));
                                } catch (std::runtime_error& e){
```

```
                driver.table = NULL;
                driver.error(e.what());
                return Parser::SYNTAX_ERR;
        }
        delete t1;
}
if (driver.table == NULL) {
        if (driver.cs != NULL) {
                delete driver.cs;
                driver.cs = NULL;
        }
} else
        try {
                if ($3) {
                        Table* t1 = driver.table;
                        WickyEngine* we = WickyEngine::getInstance();

                        driver.table = we->Project(t1, *(driver.cs));
                        driver.table->printTable();
                        delete driver.cs;
                        delete t1;
                        driver.cs = NULL;
                        t1 = NULL;
                } else {
                        driver.table->printTable();
                }
        } catch (std::runtime_error& e){
                driver.table = NULL;
                driver.error(e.what());
                return Parser::SYNTAX_ERR;
```

```
                                }


                        }
        ;
opt_all_distinct:
                        /* empty */
        |               ALL
        |               DISTINCT
        ;
selection:
                        scalar_exp_commalist { $$ = 1; }
        |               '*' { $$ = 0;}
        ;
table_exp:
                        from_clause opt_where_clause{
                        }
        ;
opt_where_clause:
                        /* empty */
        |               where_clause {


                        }
        ;
from_clause:
                        FROM table_ref_commalist {
                        }
        ;
```

4. 测试

首先检测是否能接受基础的操作。第一个为建表操作，由于已经有一张表所以提示无法创建。

```
E:\Education\My Resource\DBSD\WickyDB [master +4 ~1 -0 !]> .\wickydb.exe
Welcome to the WicyDB monitor. Commands end with ;
This is our team work. The team compose of Hai Jiewen Zhang Haiwei Yu Qiubin and Xiao Shaobin.
wickydb>select * from tab;
Table tab doesn't exist
wickydb>create table student (
------->               sno int,
------->               sname char(16) unique,
------->               sage int,
------->               sgender char (1),
------->               score float,
------->               primary key ( sno )
------->);
Table student already exists
```

下面检测 drop table 操作，在 drop table 之后可以看出数据库为空。

```
------->
------->drop table student;
wickydb>show table;
Empty database
wickydb>
------->
```

而执行 execifle 后，文件中的 sql 语句会被顺序执行

```
------->execfile small.sql
Insert: 1 wy1 21 F 66.0
Insert: 2 wy2 20 M 67.0
Insert: 3 wy3 21 F 68.0
Insert: 4 wy4 20 M 69.0
Insert: 5 wy5 21 F 70.0
Insert: 6 wy6 20 M 71.0
Insert: 7 wy7 21 F 72.0
Insert: 8 wy8 20 M 73.0
Insert: 9 wy9 21 F 74.0
Insert: 10 wy10 20 M 75.0
Insert: 11 wy11 21 F 76.0
```

执行 select 操作可以看到相应的结果

```
------->select * from student where score >= 90 and score <=95;
        sno       sname        sage     sgender       score
         25        wy25          21           F        90.0
         26        wy26          20           M        91.0
         27        wy27          21           F        92.0
```

- # IndexManager

1. 设计原则

为上层提供一个良好而又方便的 index 管理机制，能够生成、删除、获得索引，并且可以通过索引来进行 key-value 的检索、删除操作。

2. 具体设计

主要对象

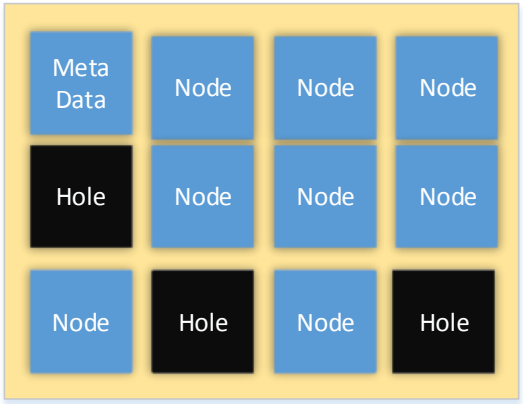
```
class Index{
protected:
        std::string name, type, fileName;
        int keyLen;
        int maxKeyNum, last;
        Node* root;
        std::list<int> holes;
        std::map<int, Node*> nodes;
}
class Node {
private:
        int keyNum;
        int parent;
        const int ptr;
        Index* index;
        bool inter;
        unsigned char * content;
}
```

动态空间分配

因为需要在磁盘上进行 B+树节点的动态分配与回收，所以需要一个动态的回收机制来管理整个磁盘空间。设计上是每个文件的第一个块记录关于当前索引的空间分配信息。包括根节点位置、索引空间范围以及所有未分配的空间。用于管理整个索引的空间分配。

而每一个节点都占有一个块的空间，你们也需要记录一些检索需要用到的信息。

所以整个索引的布局由两个部分组成，管理空间信息的首块以及存储具体索引信息的节点，这些内容共同组成了整个索引的数据库文件。这种布局的索引文件结构能够尽可能地利用磁盘空间，在封装之后能够非常方便地申请与释放磁盘空间。



空间申请：

```
Node* Index::newNode(){
        int n;
        Node* ret;
        if (holes.empty()){
                last += Block::BLOCK_SIZE;
                n = last;
        } else {
                n = holes.front();
                holes.pop_front();
        }
        BufferManager* bm = BufferManager::getInstance();
        unsigned char* buf = new unsigned char[Block::BLOCK_SIZE];
        bm->write(this->getFileName(), n, Block::BLOCK_SIZE, buf); // write block
        bm->write(this->getFileName(), n, -1); // set parent to -1
        bm->write(this->getFileName(), 0);// set keynum
        bm->write(this->getFileName(), 0);// set noninternal
        delete[] buf;
```

```
        ret = new Node(this, n);
        nodes.insert(std::map<int, Node*>::value_type(n, ret));
        return ret;
}
```

从 hole 队列中找出一个空间，在其上面生产一个 Node 并进行管理。

空间释放

```
void Index::deleteNode(Node* n){
        if (nodes.find(n->getAddr()) == nodes.end())
                throw std::runtime_error("node dosen't exist");
        holes.push_back(n->getAddr());
        nodes.erase(nodes.find(n->getAddr()));
        delete n;
}
```

将空间从 Node 上回收。

查询

```
class Key{
private:
        int len;
        unsigned char* key;
}
```

为了使键值兼容各种类型，我对其进行了一层封装，可以将所有类型的键都转换成二进制形式，方便最终的比较。

```
std::pair<Node*, int> Index::find(Key k){
        Node* C = this->root;
        if (C == NULL) return std::pair<Node*, int>(NULL, -1);
        while (C->isInternal()){
                int i = C->findV(k);
                if (i == -1) {
                        C = getNode(C->getPointer(C->getKeyNum()));
```

```
            } else if (k == C->getKey(i)){
                C = getNode(C->getPointer(i+1));
            } else {
                C = getNode(C->getPointer(i));
            }
        }
        int i = C->findV(k);
        if (C->getKey(i) == k) {
            return std::pair<Node*, int>(C, i);
        } else {
            return std::pair<Node*, int>(C, -1);
        }
}
```

扫描所有键与指针，找到相应的位置进入下个节点，若寻址到叶节点后没有找到相应的键值，则返回没找到。

插入

```
int Index::insertKey(Key K, int P){
    if (K.getLength() != keyLen)
        throw std::runtime_error("key length does not match");
    Node* L;
    if (root == NULL) {
        root = newNode();
        L = root;
    } else {
        std::pair<Node*, int> p = find(K);
        if (p.second != -1)
            return KEY_EXIST;
        L = p.first;
    }
```

```
        if (L->getKeyNum() < maxKeyNum - 1){
                L->insertInLeaf(K, P);
        } else {
                L->add(P, K);
                Node* L1 = L->split();
                L->insertInParent(L1->getKey(0), L1);
        }
        return INSERT_SUCCESS;
}
```

插入的方式为找到相应的节点，若键不存在则将键值插入。若需要将节点分裂则需要申请新的节点，并且将数据分布正确。

删除

```
void Node::deleteEntry(Key K, int P){
        deletePK(K, P);
        if (index->getRoot() == this){
                if (keyNum == 0){
                        if (!isInternal()){
                                index->setRoot(NULL);
                                return;
                        }
                        index->setRoot(index->getNode(this->getPointer(0)));
                        index->deleteNode(this);
                }
                return;
        } else if (keyNum < index->getMaxKeyNum() / 2){
                Node* Pa = index->getNode(this->parent);
                Key K1 = K;
                Node* N1;
                int pN = Pa->findV(this->getKey(0));
```

```
int pN1;


if (pN == -1){
        pN = Pa->keyNum;
        pN1 = pN - 1;
        K1 = Pa->getKey(Pa->keyNum-1);
} else {
        if (Pa->getKey(pN) == this->getKey(0)) pN ++;
        if (pN == 0){
                K1 = Pa->getKey(pN);
                pN1 = pN + 1;
        } else {
                K1 = Pa->getKey(pN-1);
                pN1 = pN-1;
        }
}
N1 = index->getNode(Pa->getPointer(pN1));

if (this->getKeyNum() + N1->getKeyNum() < index->getMaxKeyNum()){
        if (pN1 < pN){
                N1->coalesce(this, K1);
                Pa->deleteEntry(K1, this->getAddr());
                index->deleteNode(this);
        } else  {
                this->coalesce(N1, K1);
                Pa->deleteEntry(K1, N1->getAddr());
                index->deleteNode(N1);
        }
} else {
```

```
                    if (pN1 < pN){
                        N1->redistribute(this, K1);
                    } else {
                        N1->aredistribute(this, K1);
                    }
                }
            }
        }
```

删除则稍微复杂一点，如果节点需要调整，则需要做 coalasce 与 redistribute 操作。若相邻两个节点可以被结合在一起，则将他们合并并回收空间。若不能被合并在一起，则将一个节点分配到另一个节点。

3. 测试

使用 20 万条数据插入需要 5 秒钟，并且支持正确的查询与删除功能。