3130102203
余秋滨
2015年11月9日

API模块和Record模块报告

Database Implementation

一、概述

在我们的数据库实现过程中，我主要负责Record模块全部，以及API模块的部分功能。最终达到的效果是，在API层能够接收来自interpreter的语句，实现select/insert/delete语句的操作，在Record层管理记录表中数据的数据文件。主要功能为实现数据文件的创建与删除、记录的插入、删除与查找操作，并对外提供相应的接口。

二、API模块相关

在API模块中，我负责实现的函数是：

- Table* WickyEngine::Select(Table* t, Condition c)
- Table* WickyEngine::Project(Table* t, std::vector<std::pair<std::string, std::string> > cs)
- int WickyEngine::Insert(Table* t, std::vector<std::pair<std::string, std::string> > values)
- int WickyEngine::Delete(Table* t, Condition c)

这四个函数分别接收来自interpreter的语句，进行逻辑运算，然后调用底层的RecordManager，IndexManager和CatalogManager等进行数据操作。

首先是Select函数，其功能就是支持select语句。Select函数接受的输入是table变量和condition变量，table变量就是需要进行操作的表，condition变量是where后面的内容，如果condition变量为空就执行select * from XXX的命令。

```
map<string,int> opMap;
opMap["="]=0;
opMap[">"]=1;
opMap["<"]=2;
opMap["<>"]=3;
opMap["<="]=4;
opMap[">="]=5;
```

首先是使用了如上所示的一个Map，这个Map的作用是将select语句中的where条件的操作符映射到0～5的数字上，方便之后的swicht操作的判断。

```
while(!c.empty()){
      a lot of codes…
}
```

然后在定义了许多容器、变量之后，有一个大的while循环，变量c就是函数输入值condition，这是一个list容器。通过判断这个list容器是否为空来决定是否进一步进行条件判定。在这个while循环里面，要提取condition中的pair对，pair对的第一个参数是类型名称，如果是'CHAR'型的变量，要把它两边的括号去掉，所以要使用如下的循环进行操作：

```
for(itList = cond.begin(); itList != cond.end(); itList++){
      if(itList->first=="CHAR")
            itList->second = itList->second.substr(1,itList-
>second.size()-2);
}
```

完成去掉括号的操作之后，我另用一个容器把参数值保存下来：

```
for(itList = cond.begin(); itList != cond.end(); itList++){
      tempStore.push_back(itList->second);
}
```

于是tempStore里面会有三个值，第一个是被操作的attribute名字，第二个值是operation类型，第三个是判定条件，也就是类似 attribute X > 100 的这种形式。接下来从传入的table里面获取attribute列表，然后遍历这个列表，找出与tempStore[0]相等的名称的attribute，返回position位置，如果position的值在遍历以后仍为-1，说明没有找到相应的attribute，也就是说用户输入的选择语句是错误的，那么就中断了接下去的操作。这一部分的代码如下：

```
            int position=-1;
            for(int i = 0; i < tAttr.size(); i++){
```

```
                if(tAttr[i].getName()==tempStore[0]){
                position = i;
                     break;
                }
           }
```

如果正确找到了position，那么接下来就需要使用switch语句对其进行判定，这里要利用到一开始就定义好了的那个map。case语句的形式如下：

```
           case 0:
                for(int i = 0; i < resultRow.size(); i++){
                     if(type!="CHAR"){
                          float firstCmp =
atof(resultRow[i].col[position].c_str());
                          float secondCmp =
atof(tempStore[2].c_str());
                          if(firstCmp==secondCmp)
                               tempRow.push_back(resultRow[i]);
                     }
                     else{

if(resultRow[i].col[position]==tempStore[2])
                               tempRow.push_back(resultRow[i]);
                     }
                }
```

按照opMap中的定义，映射到0的操作符是等于号。由于底层Record中存储的都是string类型，因此首先要判断该attribute是什么属性的，如果是CHAR型的那么可以直接进行比较，如果不是CHAR型的那么就需要转成数字后再进行比较。如果比较成功，那么就把该行数据暂存到一个vector容器中。

接下来就是在while循环中反复执行这个过程，直到condition里面没有东西为止，因此得到的就是所有condition进行逻辑“and”之后的结果。再将这个结果进行判定，如果该行号已经做了删除标记，那么就将其从select的结果中移除即可。

project函数的思想与select函数寻找position的思想近似。project函数的接口如下：

```
Table* WickyEngine::Project(Table* t,
std::vector<std::pair<std::string, std::string> > cs)
```

第一个输入参数是table变量，即对应要操作的表。第二个输入变量是一个vector容器变量，即要project的attribute名。因此首先遍历attribute列表，将所有需要输出的attribute名字先存放在一个vector中，如下所示：

```
for(int i = 0; i < size; i++){
	string temp = attrOri[i].getName();
	flag = false;
	for(int j = 0; j < cs.size(); j++){
		if(temp == cs[i].second){
			flag = true;
			break;
		}
	}
	if(flag)
		targetNum.insert(i);
	else
		attrRes.push_back(attrOri[i]);
}
```

然后就可以用正常的遍历手段输出这些attribute的每一行信息了。

接下来说明insert函数，insert函数的接口如下：

int WickyEngine::Insert(Table* t, std::vector<std::pair<std::string, std::string> > values)

输入一个table变量，以及一个vector变量。vector变量values里存储的就是要插入的数据。当然，由上层得到的values并不一定能保证数量与table中的attribute数量一致，所以要先判断一下。

```
if(t->getAttrNum()!=values.size())
	throw std::runtime_error("The input data can't fit the table!");
```

然后要通过CatalogManager拿到table的primaryKey，代码如下：

```
CatalogManager* cm = CatalogManager::getInstance();
std::string primaryKey;
if(cm->isExist(t->getTableName())){
	Schema s = cm->get(t->getTableName());
	primaryKey = s.getPrimaryKey();
}
```

接下来构造一个values的迭代器，然后遍历values，查找这些输入值的属性，如果是CHAR的话同样要去掉首尾的单引号，代码如下：

```
     for(itr = values.begin(); itr != values.end(); itr++){

          if(itr->first=="CHAR")
               itr->second = itr->second.substr(1,itr-
>second.size()-2);
     }
```

接下来是一个for循环，在这个for循环里遍历要插入的每一个元素，检查它们是否是Primary Key，是否是Unique Key，是否存在Index，是否符合table中定义的attribute属性。首先检查类型，代码如下：

```
     if(itr->first!=attrList[countAttr].getType()){
     throw std::runtime_error("Required a "+
attrList[countAttr].getType() +" type!Insertion failed");

     }
```

其次检查是否是CHAR型，如果是的话是否符合attribute中定义的长度：

```
     if(itr->first!="CHAR")
          inputCol.push_back(itr->second);
     else{
          int attrLength = attrList[countAttr].getLength();
          if(attrLength < itr->second.size()){
               throw std::runtime_error("The string "+itr-
>second + " is too long! Insertion failed");
          }
               inputCol.push_back(itr->second);
     }
```

接着检查是否存在index，调用index的接口searchKey()即可，如果返回大于0，说明这个元素是primary key或者unique key，并且有index而且已经存在一项了，因此可以直接抛出异常：

```
          int offset=searchKey(Key thekey);
          if(offset>0){
               throw std::runtime_error("The attribute" +
attrList[countAttr].getName()+ " is unique or primary! And the
value is already exist! Insertion failed");
          }
```

如果offset<0，那么就要遍历检查一遍：

```
     if(attrList[countAttr].isUnique() ||
attrList[countAttr].getName()==primaryKey){
               for(int k = 0; k < t->rows.size(); k++){
                    if(t->rows[k].col[countAttr]==itr->second)
                         throw std::runtime_error("The
attribute" + attrList[countAttr].getName()
                              + " is unique or primary! And
the value is already exist! Insertion failed");
               }
          }
```

在以上检查都通过以后，说明这个输入的row是符合要求的，应当予以插入。因此调用RecordManager的接口进行插入即可，这将在后文介绍RecordManager的时候再描述。

接下来介绍delete函数。由于API的这一部分和Record都是我负责的，因此在编写的时候我自己把这两个地方弄的耦合度稍微高了一点，把很多本该由Record负责的操作都放在了API层。这一点是我没把握好的，自我检讨一下。

delete函数与select函数相似，同样需要一个condition的输入变量：

int WickyEngine::Delete(Table* t, Condition c);

同样需要一个map变量来映射操作符到数字0 ~ 5:

```
map<string,int> opMap;
opMap["="]=0;
opMap[">"]=1;
opMap["<"]=2;
opMap["<>"]=3;
opMap["<="]=4;
opMap[“>="]=5;
```

接着初始化一个vector，用来记录需要被delete的行号，并且初始状态是记录所有行号，也就是对应“delete from XXX”的这种情况：

```
vector<int> toDeleteIndex(t->rows.size());
for(int i = 0 ; i < toDeleteIndex.size(); i++)
     toDeleteIndex[i]=i;
```

紧接着在定义了一系列中间变量以后，与select一样，有一个while循环判断输入的condition是否为空。若不为空则进行一系列操作，首先同样要判断condition中各参数的类型，如果是CHAR就要去掉首尾的单引号：

```
          for(itList = cond.begin(); itList != cond.end();
itList++){
               if(itList->first=="CHAR")
                    itList->second = itList-
>second.substr(1,itList->second.size()-2);
          }
          tempStore.clear();
          for(itList = cond.begin(); itList != cond.end();
itList++){
               tempStore.push_back(itList->second);
          }
```

接下来与insert函数中的操作类似，进行switch操作。由于我们的index只能进行等值查找，因此只有当case=0的时候才能使用searchKey()函数来找offset，其余的情况都需要遍历整个table来确定需要delete的行号，例如case 1，也就是操作符为“>”的时候：

```
     case 1:
               for(int i = 0; i < toDeleteIndex.size(); i++){
                    string target =
rawRow[toDeleteIndex[i]].col[position];
                    if(type!="CHAR"){
                         float firstCmp =
atof(target.c_str());
                         float secondCmp =
atof(tempStore[2].c_str());
                         if(firstCmp>secondCmp)

tempIndex.push_back(toDeleteIndex[i]);
                    }
                    else{
                         if(target>tempStore[2])

tempIndex.push_back(toDeleteIndex[i]);
                    }
               }
```

于是在一连串的switch之后确定了最后要delete的行号，将其置入table的deletedIndex容器内，做假删除操作。

三、Record模块相关

Record模块是直接对表进行操作的地方，首先是tuple的声明如下：

```
class Tuple{
public:
     std::vector<std::string> col; //each column
     Tuple(){}
     Tuple(std::vector<std::string> v){
          col = v;
     }
};
```

接着是表的声明如下：

```
class Table{
private:
     std::string tableName;              //the name of table
     int attrNum;                  //the number of
attributes(column)
     std::vector<Attribute> attrList;   //the list of
attributes
     int tailOffset;

public:
     std::vector<Tuple> rows;            //the data of the
table
     std::set<int> deletedIndex;
     Table(std::string tableName);
     bool CreateTable(std::vector<Attribute> attrList);     //
initial the table with attributes list

     int getAttrNum(){return attrNum;}                      //
return the number of attributes
     // int getTableLen(){return length;}
     std::string getTableName(){return tableName;}          //
return the table name
     void printTable();                      //print all the
rows of table

     std::vector<Attribute> getAttrList(){return attrList;}
     int getTailOffset(){return tailOffset;}
     void setTailOffset(int x){tailOffset=x;}
};
```

另外在schema里面定义了attribute：

```
class Attribute
{
private:
     std::string attrName;
     std::string type;
     int length;
     std::string index;
     bool unique;
    friend class Schema;

public:
     Attribute(){};
     Attribute(std::string name, std::list<std::string>
properties);
     ~Attribute(){};
     std::string getName(){return attrName;}
     std::string getType(){return type;}
     int getLength(){return length;}
     std::string getIndex(){return index;}
     bool isUnique(){return unique;}
};
```

所以总体数据存储结构是，table里面存储了一个attribute的容器，存储了一个Tuple的容器，另外还存储了一个判断是否删除的容器。attribute容器中存放着一系列的attribute对象，这些对象包含了attribute的名字、类型、是否有index、是否Unique、长度等等信息。而在Tuple里面有一个col的容器，这里面存储的就是一行数据。deletedIndex容器里存储的就是假删除的标记。

接下来是RecordManager的接口：

```
class RecordManager{
public:
     int insertTuple(Table* table, Tuple tuple, int offset);
     int deleteTuple(Table* table, int offset);
     std::vector<Tuple> selectTuple(Table* table,
std::vector<int> offset);
     Table readTable(Schema s, BufferManager *b, int
offset=0);
     bool writeTable(Table table, BufferManager *b, int
offset=0);
     void deleteTable(std::string tableName, BufferManager
*b);
```

```
    void Split(std::string src, std::string separator,
std::vector<std::string>& v);
};
```

RecordManager直接与BufferManager交互的是writeTable，readTable和deleteTable函数，在writeTable中，会空出文件的第一个block，这个block不用来写数据，而是用来写这个table的各种信息，诸如有多少attribute、有多长等，结构图可以用右图来表示。

write函数中的offset是需要更新的起始偏移地址，默认为0，也就是从第二个block开始写入所有数据。另外，在写入数据的时候，会将deletedIndex写在每一行的开始，0代表正常数据，1代表已删除的数据：

```
    for(int i = 0; i < table.rows.size(); i++){
        for(int j = 0; j < table.rows[i].col.size(); j++){
            output = output + " "+ table.deletedIndex[i] +"
"+ table.rows[i].col[j];
        }
    }
```

而后将table的基本信息，attribute的数量以及table长度等信息写入第一个block，然而根据偏移量将数据写入数据区：

```
    b->write(filename,Block::BLOCK_SIZE+offset, output);
    b->write(filename,0,(int)output.size());
```

readTable就是writeTable的逆过程，readTable函数需要一个传入的Schema参数，Schema参数中保存了table的元信息。然后readTable会根据第一个block的信息来判断文件是否正常：

```
    Split(raw, " ", rawVec);
    int attrNumber = atoi(rawVec[0].c_str());
        if (attrNumber != result->getAttrNum())
        std::cout<<"WARNING! the data is
unsafe!"<<std::endl;
```

在读入第一个block的数据后，用Split函数按照空格进行分词，第一个字符串代表的是attribute的数量，如果与Schema传入的信息不符合的话就会作出警告。

接下来将会根据offset读取对应的block，如果没有指定offset则读取全部的block数据，处理后存入deletedIndex和rows中。

其中用到的Split函数就是用来分词的，可以根据输入的字符来进行任意分词，核心处理循环如下：

```
 do
{
    index = str.find_first_of(separator,start);
    if (index != std::string::npos)
    {
        substring = str.substr(start,index-start);
        dest.push_back(substring);
        start = str.find_first_not_of(separator,index);
        if (start == std::string::npos) return;
    }
}while(index != std::string::npos);
```

其中str就是输入的字符串，seperator就是输入的分隔字符。

deleteTable()函数较为简单，直接调用了BufferManger的接口即可。首先判断要删除的表是否存在，然后进行删除即可：

```
 if(!b->isFileExists(tableName)){
      cout<<"The table "<<tableName<<" isn't
exist!"<<endl;
      return;
 }
 b->removeFile(tableName);
```

而insertTuple、deleteTuple、selectTuple的功能也很明了，就是根据输入的table，offset等信息对相应的Tuple进行操作，与API进行交互。

四、总结

本次大程让我学到了很多东西，对于如何实现一个数据库有了直观的认识。也深刻的明白了设计和实现一个数据库是一件需要仔细推敲、耐心实践的事情。最终我写的模块能和组内其他人的模块合并起来运行，让我十分高兴，但是我也意识到我仍然有一些地方值得改进，例如应当降低API和Record之间的耦合度等等。