Record Manager 设计报告

3120101973 陆洲

一、模块概述

MiniSQL 的设计要求中需要实现创建记录、查找记录和删除记录的功能。

查找记录:

用户可以通过指定用 and 连接的多个条件进行查询，支持等值查询和区间查询。

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。

RecordManager 是集中管理数据库中每条记录的模块，是上层 API、Interpreter 和底层 BufferManager、DBFile 之间的沟通渠道。它不直接读取数据，通过 BufferManager 请求并接收 DBfile 中的记录，同时也对外提供相应接口。

二、主要功能

Record Manager 负责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（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。其中记录的查找操作要求能够支持不带条件的查找和带一个条件的查找（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.选择语句：

该语句的语法如下：

select * from 表名 ; 或 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：=<> < > <= >=

若该语句执行成功且查询结果不为空，则将结果返回给 Interpreter：第一行为属性名，其余每一行表示一条记录；若查询结果为空，则返回 NULL。

2.插入记录语句：

该语句的语法如下：

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

若该语句执行成功，则返回成功得到的信息；若失败，则返回 NULL。

3.删除记录语句

该语句的语法如下：

delete from 表名 ;或 delete from 表名 where 条件 ;

若该语句执行成功，则返回成功得到的信息，其中包括删除的记录数；若失败，则返回 NULL。

三、主要对外接口

1.读取记录，用于选择语句、索引查找
CArray<record, record&>* readRecords
(CString tableName, TableInformation tableInfo, CArray<Compare, Compare&>* cp);
2.存储记录，用于插入记录语句
void storeRecord(CString tableName, Record data);
3.删除记录，用于删除记录语句
void removeRecord
(CString tableName, TableInformation* tableInfo, CArray<Compare, Compare&>* cp)

四、设计思路

由于在 DBFile 的内部，每一个属性虽然没有完整的 Record 易于管理，但由于有时只需要返回某几个特定属性，为保证其灵活性，数据是通过 Array 的形式存取的，同时因为数组大小是不定量，因此使用动态分配的方式。

在上层，我们却是通过 Record 的形式显示输出，因为一条 Record 必须被看作是一个整体，有很多操作是对一条 Record 的所有属性进行的。因此需要在 RecordManager 内部设计两个互相转换的函数，即 recordToArray 和 arrayToRecord，主要方便上层输入与下层各个 Manager 进行沟通。

在 Interpreter 解读分析指令，调用 RecordManager 的接口，如果要读取，就先在 Buffer 中寻找，如果没有，再到 DBFile 中读取，在经过转化成 Record 之后再输出；如果要添加或删除就将用户输入的 Record 转为 Array 形式。

在读取底层文件时，由于数据量并不是很大，我们采取了最简单的线性查找方法（linear search），这种方法的时间开销是 $t_S + b_r * t_T$。

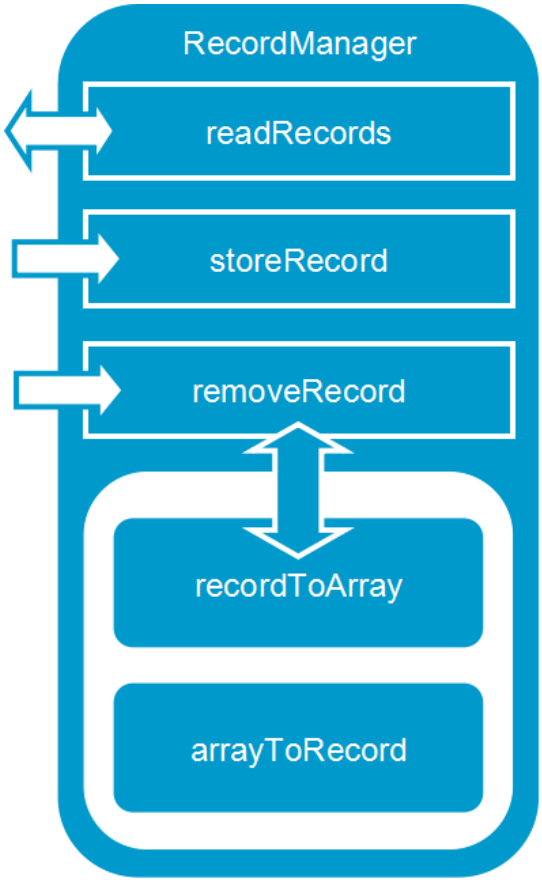
五、整体架构

在模块中，将调用独立的基类模块 Basic，该模块中有 MiniSQL 使用的 Record 类、TableInformation 类的定义。

整个模块包括 5 个函数，两个内部格式转换函数，三个分别负责对 Record 进行读取、存储、删除的对外接口。在进行 Record 读取的过程中，表格所含的信息也会发生变化（即表格的 metadata），因此被操作的表格对应的 TableInformation 等也会被更改。

整体模块接收 Array 作为输入，Record 作为输出，若进行了不允许的操作，比如空表删除等，也会及时将错误返回给 Interpreter。

模块设计图如下：



六、关键函数及伪码

1.读取记录

```
CArray<Record, Record&>*
RecordManager::readRecords(CString tableName, TableInformation tableInfo,
CArray<Compare, Compare&>* cp)
{
    if(cp == NULL)//如果是无比较条件的选择语句走这个分支
    {
        //动态分配内存
        .......
        bmp->findTableEntry(tableName, add);
        if(add.blockAddress == -1 || add.indexAddress == -1)
        {//在文件里查找不到这个记录的块或索引地址，返回 NULL
            ......
            return NULL;
        }
        //动态分配地址
        ......
        add = bmp->nextRecordPosition(add);
        if(add.blockAddress != -1 && add.indexAddress != -1)
```

```
        {//文件里存在记录的块及索引地址
            while(add.blockAddress != headadd.blockAddress || add.indexAddress != headadd.indexAddress)
            {//已知要读取的首尾地址，从首地址指向的记录一直读到尾地址
                content = bmp->readData(add,size);
                p = arrayToRecord(content, tableInfo);
                p->Address = add;
                rp->Add(*p);
                //delete [] content;
                content.erase();
                add = bmp->nextRecordPosition(add);
            }
            return rp;
        }
        else
        {
            //若发生地址错误，返回 NULL
        }
    }
    else//有条件语句的分支
    {
        //动态分配内存
        .......
        bmp->findTableEntry(tableName, add);
        if(add.blockAddress == -1 || add.indexAddress == -1)
        {//在文件里查找不到这个记录的块或索引地址，返回 NULL
            ......
            return NULL;
        }
        //动态分配地址
        ......
        add = bmp->nextRecordPosition(add);
        if(add.blockAddress != -1 && add.indexAddress != -1)
        {//文件里存在记录的块及索引地址
            while(add.blockAddress != headadd.blockAddress || add.indexAddress != headadd.indexAddress)
            {//从首地址一直读到尾地址
                ......
                //在读取每一条记录的同时也进行条件比较
                for(i = 0; i < cp->GetSize(); i++)
                {
                    attributeValue av = p->value[(*cp)[i].item1];
                    if(!(*cp)[i].statisfy(av))
                        break;
```

```
                    }
                    if(i == cp->GetSize())
                        rp->Add(*p);

                    delete [] content;
                    add = bmp->nextRecordPosition(add);
                }
                return rp;
            }
            else
            {
                //若发生地址错误，返回 NULL
            }
        }
}
```

读取记录主要为选择语句服务，上述代码区分了条件查询和普通查询，其中动态分配 Compare 类使得 MiniSQL 可以操作多条件查询。上述代码对可能出现的错误即空表查询（访问不存在的地址）也进行了防止。

2.存储记录

```
void RecordManager::storeRecord(CString tableName, Record data)
{
    String head;
    int size;
    head = recordToArray(data, size);
    bmp->addRecord(tableName, head, size);//此处调用 BufferManager 已有的方法即可
    delete [] head;
}
```

存储记录主要用于添加记录，其中可能出现的错误是重复添加，这一点在 Interpreter 里面已有实现，因此不重复实现。

3.删除记录

```
void RecordManager::removeRecord(CString tableName, TableInformation* tableInfo,
CArray<Compare, Compare&>* cp)
{
    CArray<Record, Record&>* rp;
    if(cp == NULL)//无条件删除走以下分支
    {
        Address headAdd, add, nextAdd;
        bmp->findTableEntry(tableName, headAdd);
        if(headAdd.blockAddress == -1 || headAdd.indexAddress == -1)
        {
            //若文档中不存在该记录，跳出该函数，返回错误
            ……
        }
        else
```

```
        {//若文档中存在该记录，从首地址一直进行删除操作至尾地址
            ……
            if(add.blockAddress != -1 && add.indexAddress != -1)
            {
                while(headAdd.blockAddress != add.blockAddress
                    || headAdd.indexAddress != add.indexAddress)
                {//若在 Buffer 里面找不到记录，则取 Disk 里读取
                    int i = bmp->readBlockFromDisk(add.blockAddress);
                    ……
                    bmp->buffer[i].removeData(add.indexAddress);
                    add = nextAdd;
                }
                bmp->rmTableEntry(tableName);
            }
            else
                bmp->rmTableEntry(tableName);
        }
    }
    Else//有条件删除走以下分支
    {
        rp = readRecords(tableName, *tableInfo, cp);
        if(rp != NULL)
        {
            for(int i = 0; i < rp->GetSize(); i++)
            {
                bmp->rmRecord((*rp)[i].address);
            }
            delete rp;
        }
    }
}
```

上述函数区分了条件删除和一般删除，且通过动态分配 Compare 类使得多条件查询后的删除可以操作。对于可能出现的空表删除也进行了防止处理。