Minisql个人报告

郑天杨 3160102142

1 Interpreter

1.1 Minisql语法

解释器解析并执行SQL语句。支持如下语法：

表建立

```
CREATE TABLE tbl_name (create_definition,...)

create_definition:
    col_name column_definition | PRIMARY KEY(col_name)

column_definition:
    data_type [UNIQUE]

data_type:
    INTEGER | FLOAT | CHAR(M)
```

单条记录的长度应小于页长。支持单属性的主键定义。

表删除

```
DROP TABLE tbl_name
```

因为没有外键，所以不需要级联删除。

索引建立

```
CREATE INDEX index_name ON tbl_name (col_name,...)
```

建立索引的方式为B-Tree。如果不指定索引名，索引名默认和变量名一致。

索引删除

```
DROP INDEX index_name ON tbl_name
```

记录插入

```
INSERT INTO tbl_name VALUES (value_list)

value:
    {expr | DEFAULT}

value_list:
    value [, value] ...
```

重复时报错。

记录删除

```
DELETE FROM tbl_name [WHERE where_condition]

where_condition:
    value_operation [AND value_operation] ...

value_operation:
    col_name {= | >= | <= | > | < | <>} value
```

记录查找

```
SELECT
    select_expr [, select_expr ...]
    [FROM table_references
    [WHERE where_condition]]

where_condition:
    value_operation [AND value_operation] ...

value_operation:
    col_name {= | >= | <= | > | < | <>} value
```

执行脚本文件

```
EXECFILE file_name
```

1.2 解释器实现

解释器shell是在`Shell.cpp`中的一个while循环。

```
    //initialize database

    while (true) {
        //get user input
        //store in tokens[]
        if (tokens[0] == "execfile") {
```

```
            if (tokens.size() == 1) //error
            else {
                ifstream infile(tokens[1]);
                while (!infile.eof()) {
                    //get user input
                    //parse
                    //execute
                    //show result
                }
            }
        } else {
            //get user input
            //parse
            //execute
            //show result
        }
    }
```

读取用户输入后，解释器首先解析字符串。在`StringUtils.cpp`中定义了系列词法分析的辅助函数。解析结果存储在`vector<string> tokens`容器中。

然后进行语法分析。语法分析的结果储存在`Command`类的一个子类中。

```
class Command {
private:
    double total_time;
    clock_t start, end;
public:
    Command();

    virtual Result execute(DbInterface &db) = 0;

    virtual ~Command();

    friend ostream &operator<<(ostream &os, const Command &command);

};
```

以`CreateTable`为例，它继承`Command`类并重写`execute`函数。语法分析的结果存在私有变量中。

```
class CreateTable : public Command {
    friend class TableMetaPage;
private:
    string Ddl = "";
    string tableName = "";
    string primaryKey = "";
    vector<column_t> columns; //column defined in Config.h
```

```
public:
    CreateTable();

    const string &getDdl() const;

    void setDdl(const string &ddl);

    const string &getTableName() const;

    void setTableName(const string &tableName);

    const string &getPrimaryKey() const;

    void addColumn(string columnName, string dataType, int charLength, bool
isUnique);

    void addPrimaryKey(string column);

    Result execute(DbInterface &db) override;

    friend ostream &operator<<(ostream &os, const CreateTable &table);

};
```

语法分析在`Parser.cpp`中完成。伪代码如下，解析后返回一个`unique_ptr<command>`指针。

```
unique_ptr<Command> Parser::parse(string sqlString)
{
    try {
        if (tokens[0] == "create" && tokens[1] == "table") {
            ///create table
            return unique_ptr<Command>(createTable);
        } else if (tokens[0] == "drop" && tokens[1] == "table") {
            ///drop table
            return unique_ptr<Command>(dropTable);
        } else if (tokens[0] == "create" && tokens[1] == "index" && tokens[3]
== "on") {
            ///create index
            return unique_ptr<Command>(createIndex);
        } else if (tokens[0] == "drop" && tokens[1] == "index") {
            ///drop index
            return unique_ptr<Command>(dropIndex);
        } else if (tokens[0] == "insert" && tokens[1] == "into" && tokens[3] ==
"values") {
            ///insert
            return unique_ptr<Command>(insert);
        } else if (tokens[0] == "delete" && tokens[1] == "from") {
            ///delete
            return unique_ptr<Command>(del);
```

```
        } else if (tokens[0] == "select") {
            ///select
            return unique_ptr<Command>(select);
        } else {
        }
    } catch (...) {
        cerr << "Syntax error!" << endl;
    }

    return unique_ptr<Command>(new Unknown);

}
```

2 Record

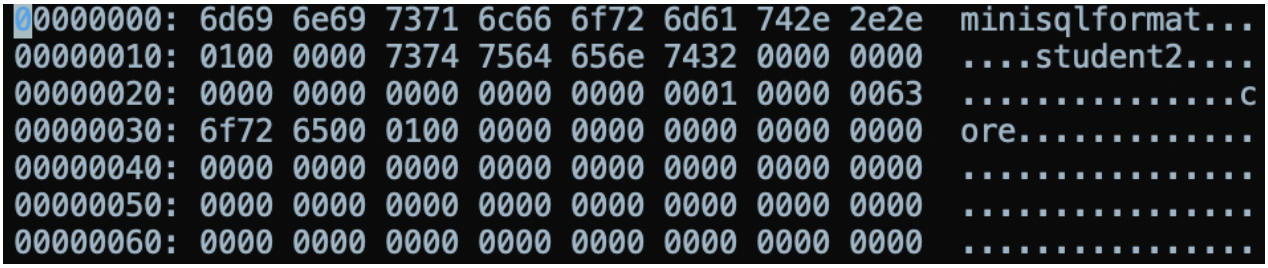
2.1 记录设计

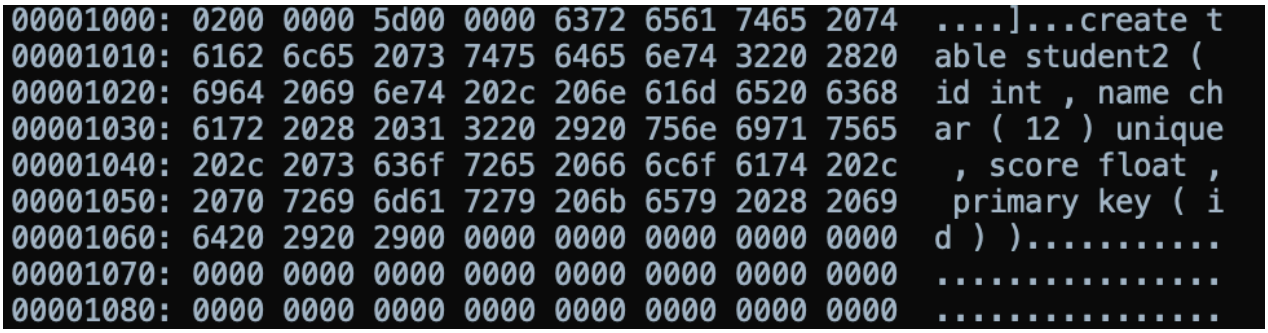
记录格式的设计是和其他部分的功能紧密关联的。这里给出概述。

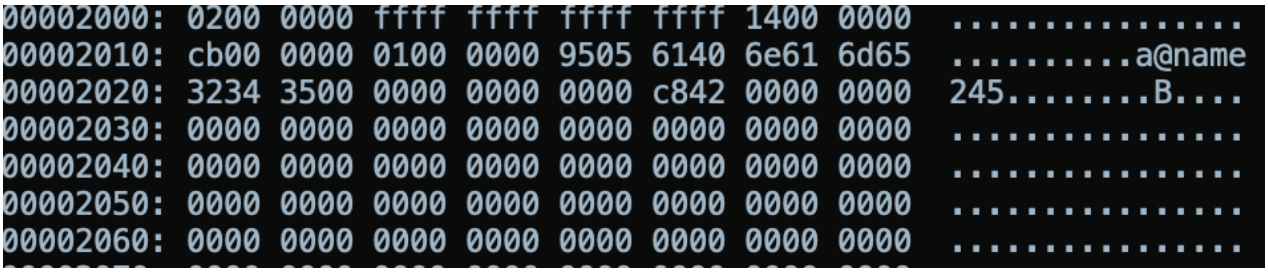
记录存储在单个磁盘文件中，文件名在`Shell.cpp`中指定。文件分成长度为`4096`的页，不支持跨页存储。下表描述了不同页的位置和功能。

Page type	ID	描述
DbMetaPage	0	存储数据库元信息。这包括表的个数、索引个数、表元信息所在位置、索引元信息所在位置等。
TableMetaPage	存储在DbMetaPage中	存储表的元信息。包括表名、DDL、第一个数据页的ID等。
TablePage	第1页存储在TableMetaPage中，其他页可以通过前一页或后一页找到	存储记录。包括记录个数、bitmap、记录等。
IndexMetaPage	存储在DbMetaPage中	存储索引的元信息。与TableMetaPage类似。
IndexPage	第1页存储在IndexMetaPage中，其他页可以通过前一页或后一页找到	存储索引。与TablePage类似。

下图用二进制格式显示了DbMetaPage、TableMetaPage、TablePage开头。







2.2 磁盘管理

直接读写磁盘记录的函数在`DiskManager.cpp`中。主要有以下函数：

```
    void writePage(page_id_t pageId, const char *pageData);

    void readPage(page_id_t pageId, char *pageData);

    page_id_t allocatePage();
```

`writePage`用于向页中写数据。pageData的类型是`char[4096]`。

`readPage`用于读页中的数据。

`allocatePage`用于设置一个新的页。

读和写都用fstream完成。

```
private:
    fstream dbIo;
```

3 Buffer

Buffer是存储管理器中的缓冲池。 它负责将物理页面从主内存来回移动到磁盘，允许DBMS支持大于系统可用内存量的数据库。其操作对系统中的其他部分是透明的。例如，系统使用其唯一标识符（page_id）向缓冲池请求页面，并且它不知道该页面是否已经在内存中，或者系统是否必须从磁盘检索它。

3.1 内存中页的定义

内存中页定义为Page类：

```
class Page {
    friend class BufferPoolManager;
    friend class DbInterface;
    friend class DbMetaPage;
    friend class TableMetaPage;
    friend class TablePage;

private:
    bool isDirty = false;
    page_id_t pageId = INVALID_PAGE_ID;

protected:
    char data[PAGE_SIZE]{};

public:
    page_id_t getPageId() const { return pageId; }

    void resetMemory() { memset(data, 0, PAGE_SIZE); }

};
```

它本质上是`char[4096]`、页序号和脏页标记。不同类型的页定义为Page的子类。以TableMetaPage为例：

```
class TableMetaPage : Page {
    friend class DbInterface;

private:
    string ddl = "";
    string tableName = "";
    string primaryKey = "";
    vector<column_t> columns; //column defined in Config.h

    int tupleLength;

    page_id_t rootId;

public:
```

```
    explicit TableMetaPage(Page *p);

    void composePage();

    void composePage(page_id_t rootId, string ddl);

    void parsePage();

    const string &getDdl() const;

    int getTupleLength() const;

    int calMaxNumOfTuples();

    page_id_t getRootId() const;

    int getAttrOffset(string attrName) const;

    int getAttrSize(string attrName) const;

    string getAttrType(string attrName) const;

    bool isAttrUnique(string attrName);
};
```

TableMetaPage继承Page类。除了存储数据，还定义了对解析和组装页的方法。调用parsePage()后，将data中的数据解析为私有变量。调用composePage()后，将私有变量中的数据写入data。

3.2 调页算法LRU

Least Recently Used算法用一个模版类实现。

```
template <class T>
class LruReplacer {
private:
    vector<pair<T, int> > tracker;
    int lruNo = 0;

public:
    LruReplacer();

    virtual ~LruReplacer();

    void insert(const T &value);

    bool victim(T &value);

    bool erase(const T &value);
```

```
    int size();

    const vector<pair<T, int>> &getTracker() const;

};
```

它本质上是一个记录`T`被创建时间的表。`vector<pair<T, int> > tracker`用于记录`T`被插入时的序号，越早使用序号越小。当需要置换时，函数`victim`返回序号最小的`T`。

在使用时，将模版类实例化为`LruReplacer<Page *>`。

3.3 页表

以下是缓冲区管理类BufferPoolManager的定义，其中包含了页表的定义：

```
class BufferPoolManager {
    friend class Page;
private:
    int poolSize;
    Page *pages; //pages in memory of number PAGE_SIZE
    DiskManager *diskManager;
    LruReplacer<Page *> *replacer;
    map<page_id_t, Page *> pageTable; //for quickly find page in pages
    list<Page *> *freeList;

public:
    BufferPoolManager(int poolSize, DiskManager *diskManager);

    virtual ~BufferPoolManager();

    /* new page ID is stored in pageId */
    Page *newPage(page_id_t &pageId);

    bool deletePage(page_id_t pageId);

    Page *fetchPage(page_id_t pageId);

    bool flushPage(page_id_t pageId);

};
```

空页使用List容器管理，定义为`list<Page *> *freeList`。每次启动数据库shell时，初始化freeList，为一定数目（这里设定为1000）的Page分配内存。

存有数据的页使用Map容器管理，定义为`map<page_id_t, Page *> pageTable;`。它是一个哈希函数，将页号映射为Page的指针。

3.4 缓冲区管理

`newPage`分配一个新页。当需要调页时，先检查freeList中是否有空页。如果有，则使用其中的空页；如果没有，则使用LruReplacer换页。如果被换的页面为脏页，需将其写入磁盘。另外，分配新页时还将其标记为脏页。

`fetchPage`返回Page指针。它首先在页表中查找，如果找到则返回。如果页表中没有，则说明该页尚未调入内存，于是从磁盘中读取。从磁盘中读取的步骤和分配新页基本一致。

因为没有实现并发，所以这里不需要pin或unpin页。

4 Catalog

Catalog处理元数据。

4.1 数据库元数据

数据库元数据的格式如下：

```
/**
* | HEADER (16) | record_count (4) | entry_1_name (32) | entry_1_meta_id (4) |
...
*/
```

HEADER用于标志Minisql格式。紧接着存储数据库表、索引元数据的名称和page ID。

在DbMetaPage类中，可以将记录读为

```
    map<string, int> entries;
```

并进行相应的增删操作。

4.2 表元数据

表元数据的格式如下：

```
/**
 * | root_id (4) | ddl_size (4) | CREATE TABLE … ;
 */
```

在磁盘上只存储DDL。启动数据库后，若需要对表进行操作，则解析DDL，解析结果存在TableMetaPage类的私有变量中。

```
private:
    string ddl = "";
    string tableName = "";
    string primaryKey = "";
    vector<column_t> columns; //column defined in Config.h

    int tupleLength;

    page_id_t rootId;
```

5 Interface

API是数据库的编程接口。它根据`Parser`的解析结果执行命令，使用`Diskmanager`和`BufferManager`管理磁盘与内存，并返回执行结果。定义如下：

```
class DbInterface {
private:
    DiskManager *diskManager;
    BufferPoolManager *bufferPoolManager;
    map<string, int> tableStart;
    map<string, int> tableEnd;
    DbMetaPage *dbMetaPage;

public:
    DbInterface();

    ~DbInterface();

    void init(string name);

    void writeTableMeta(const string &tableName, string data);

    void deleteTableMeta(const string &tableName);

    void insertTuple(const string &tableName, vector<value_t> tuple);

    vector<vector<string>> readTuple(const string &tableName, bool selectAll,
                                     vector<string> columnNames,
vector<whereClause> wheres);

    void createIndex(const string &tableName, const string &attrName, string
data);

    int readIndex(const string &tableName, const string &attrName, int mode,
Data &key, int val);

    void deleteIndexMeta(const string &tableName, const string &indexName);
```

```
private:
    void readTable(page_id_t rootId, const string &type, int attrSize, int
attrOffset, vector<Data> &keys);

    Tree readTree(page_id_t metaId, const string &type, int attrSize,
map<page_id_t, TreeNode *> &id);

    int adjustIndex(Tree T, int mode, const Data &key, int value, const string
&type, map<page_id_t, TreeNode *> id);
};
```

启动数据库时，调用init()初始化diskManager，bufferPoolManager等。

5.1 CREATE TABLE

调用writeTableMeta()。创建新的TableMetaPage和TablePage。CREATE TABLE存储DDL，在启动数据库时重新将其解析为表元信息。

5.2 DROP TABLE

调用deleteTalbeMeta()删除表。

5.3 INSERT

调用insertTuple()。INSERT首先检查约束，在没有冲突的情况下找到表的最后一页并插入数据。

5.4 SELECT

调用readTuple()。SELECT和where子句结合从数据库中取回信息。

5.5 DELETE

调用deleteTupl()。在删除记录时采用后面记录覆盖前面记录的方式。