MiniSQL设计报告

> 人员分工
>
> 孙恺元：数据结构设计、Interpreter
>
> 石蒙：文件结构设计、Record Manager、Catalog Manager、Buffer Manager、API
>
> 杨琛：Index Manager、BpTree

1 需求描述

本实验要求设计一个简单的DBMS，使得用户可以通过命令行界面输入SQL语句实现表的建立与删除，索引的建立与删除，以及记录的增删改查。

1.1 功能需求

本数据库管理系统需要实现以下功能。

- 用户通过命令行输入SQL语句，DBMS通过解释语句，实现对应的功能。
- 用户通过命令行获得SQL语句的执行结果，包括成功执行信息、详细出错信息。
- 实现表的存储，将其保存在磁盘上。
- 实现缓冲区，负责数据块的读入和写出；实现缓冲区替换算法，提高执行效率。
- 实现表的属性的B+树索引。
- DBMS提供6种查找条件，分别是= <> < > <= >=
- 可以运行SQL脚本。

1.2 SQL格式需求

SQL语句的总体格式需求包括：

- 可以分行输入一条SQL语句。
- 每条语句结尾必须有分号。
- 所有关键字全部小写。

1.2.1 创建表

格式：

```
create table tableName (
    列名 类型 ,
    列名 类型 ,
    ......
    列名 类型 ,
    primary key ( 列名 )
);
```

实例：

```
create table student (
        sno char(8),
        sname char(16) unique,
        sage int,
        sgender char (1),
        primary key ( sno )
);
```

1.2.2 删除表

格式：

```
drop table 表名 ;
```

实例：

```
drop table student;
```

1.2.3 创建索引

格式：

```
create index 索引名 on 表名 ( 列名 );
```

实例：

```
create index stunameidx on student ( sname );
```

1.2.4 删除索引

格式：

```
drop index 索引名 ;
```

实例：

```
drop index stunameidx;
```

1.2.5 查找记录

格式：

```
select * from 表名 where 条件 ;
```

实例：

```
select * from student;
select * from student where sno = '88888888';
select * from student where sage > 20 and sgender = 'F';
```

1.2.6 插入记录

格式：

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

实例：

```
insert into student values ('12345678','wy',22,'M');
```

1.2.7 删除记录/全表

格式：

```
delete from 表名 where 条件 ;
```

实例：

```
delete from student;
delete from student where sno = ‘88888888’;
```

1.2.8 退出

格式：

```
quit;
```

1.2.9 执行SQL脚本

格式：

```
execfile 文件名;
```

实例：

```
execfile file.txt;
```

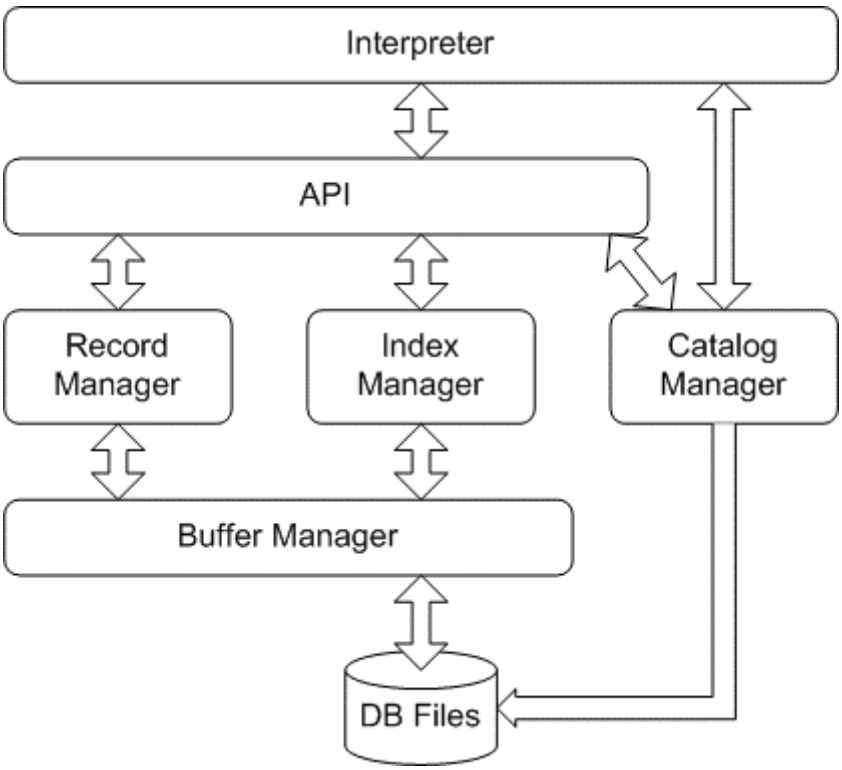
1.3 数据需求

对数据的需求如下：

- 数据类型：int，float，char三种，其中char后接一括号，表示其长度，值在[0, 255]
- 表：一共支持最多32个属性，每个属性可以指定是否为unique
- 索引：只能对unique属性做索引
- 查找：提供and查找

2 系统结构

MiniSQL系统体系结构如下图



各模块设计如下。

2.1 Interpreter

该模块是直接与用户交互，负责读入用户输入的字符串，判断其语法合理性。在判断语法正确后，将语句解释成不同的数据，传给API和Catalog Manager模块以供使用。该模块将接受查询记录的返回结果，并显示所有查询得到的语句；将接受删除记录的返回结果，并显示删除的记录数量。该模块是所有抛出异常的最顶层，并且会显示异常信息。

2.2 API

API模块接受Interpreter解释后的数据，并且从Catalog Manager中获得相应的表的信息，然后调用相应的下层模块的功能函数。

2.3 Catalog Manager

该模块负责管理数据库的模式信息，包括

- 数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。
- 表中每个属性的定义信息，包括属性类型、是否唯一等。
- 数据库中所有索引的定义，包括所属表、索引建立在哪个字段上等。

该模块负责提供上述操作的接口，供上层和下层模块使用。注意到，为了减少模块之间的耦合度，在解释器中得到的数据可以直接调用该模块的函数，以更新数据库的总体信息。

2.4 Record Manager

该模块负责管理表中的数据，主要包括增删改查，其中查询需要支持等值、不等值、开闭区间查找。

数据需要读入内存来处理，处理的大小最小单位是缓冲区块大小。

该模块接受API传进来的待处理的数据，并向Buffer Manager请求数据块。

2.5 Index Manager

该模块负责实现某表的某属性的B+树索引的建立、删除等，同样支持增删改查。

B+树中的节点大小与缓冲区块大小相同，叉数由节点大小和索引键大小得到，具体计算方法在详细设计报告中会具体介绍。

2.6 Buffer Manager

该模块负责磁盘和内存的交互，可以根据需要读取磁盘中的数据进内存、将缓冲区中的数据写回磁盘。该模块是所有Manager处理数据时必须交互的模块，因为其他Manager模块的数据均是从该模块读取，不直接接触磁盘。

为了提高效率，设计LRU替换策略、脏位和锁定。设计一个缓冲区块的大小为4KB。

3 详细设计

3.1 数据结构

我们自底向上地描述一个表的数据结构。

3.1.1 Data及其子类

考虑到我们一共需要实现三个数据类型，分别是整数、单精度浮点数、字符串，因此我们无法用一个类来实现。模板类无法解决多类型的问题，因为对字符串来说，需要特殊的标记和信息。

为此，我们设计了Data类及其三个子类，iData、fData和sData。

Data类定义如下

```
class Data
{
public:
    int type;//0:int;1:float;otherwise vchar
};
```

Data类仅包含一个类型。

三个子类定义如下

```
class iData : public Data
{
public:
    int value;

    explicit iData(int x) : Data(), value(x)
    {
        type = 0;
    }
};

class fData : public Data
{
public:
    float value;

    explicit fData(float x) : Data(), value(x)
    {
        type = 1;
    }
};
```

```
class sData : public Data
{
public:
    std::string value;

    explicit sData(std::string x) : Data(), value(std::move(x))
    {
        type = x.size() + 2;//为了防止和0 1冲突
    }
};
```

其中iData和fData直接包含了一个值value，并且设置了父类的type；对sData来说，还需要一个表示长度的量。很遗憾，由于字符串本身的长度在0~255之间，而0和1已经用来表示int和float，因此我们无法使用一个char来即表示三个类型又表示长度。解决方法是使用int类型的type，对字符串来说，其type的值是长度+2，避免了与0 1冲突（这里我们承认了0长度字符串的存在）。

在实际使用时，我们充分利用了多态机制，在任何接口都使用Data*指针，避免了重复劳动。需要获得值时，我们用如下方法

```
//假设有一个Data* d
if (d.type == 0) //int
    int value = ((iData *)d).value;
```

自顶向下的强制转化一定是合法的。

3.1.2 Tuple

Tuple类表示一个元组，由若干个字段组成，包含一个Data*的向量和一个address的int变量，其中address用来表示该元组所在的记录文件的偏移地址。

```
class Tuple
{
private:
    std::vector<Data *> data;
    int address;
public:
    explicit Tuple(std::vector<Data *> d) : data(std::move(d))
    {}
};
```

3.1.3 Attribute

Attribute结构表示一个属性的信息

```
/* 属性 */
struct Attribute
{
    std::string name;   // 属性名(不超过80个字符)
    byte type, length;  // 类型：type=0 表示 int，type=1 表示 float，type=2 表示
char（length为char的长度）
    bool isUnique;      // 是否是唯一的
    bool notNULL;       // 是否不能为空
};
```

属性名不超过80个字符，另有两个字节，分别表示类型和字符串长度（因为我们无法用一个字节表示257个数字），此外还有唯一和非空两个布尔值。

3.1.4 Index

```
/* 索引 */
struct Index
{
    std::string name;  // 索引名(不超过80个字符)
    byte indexNum;     // 索引在第几个属性上，Index和它所在的表有关，不能与表分开使用
};
```

Index结构记录了属性存在于表的第几个属性上，因此Index类能且只能在Table类里使用。

3.1.5 Table

有了以上几个类和结构，就可以构建表了。

```
/* 表 */
class Table
{
public:
    std::string name;           // 表名
    byte attrCnt;               // 属性数量
    byte indexCnt;              // 索引数量
    byte primary;               // 主键在第几个属性上，-1也就是255表示没有主键
    Index index[32];            // 最多32个索引，从0开始
    Attribute attr[32];         // 最多32个属性，从0开始
    std::vector<Tuple *> tuple; // 所有的元组
};
```

3.1.6 COMPARE

COMPARE是一个枚举，用于where子句查询，提供了6种比较符号，分别是等于、不等于、大于、小于、大于等于、小于等于

```
typedef enum
{
    e, ne, gt, lt, ge, le
} COMPARE;
```

3.1.7 WhereQuery

专门用于where子句查询的一个结构

```
struct WhereQuery
{
    explicit WhereQuery(Data &d) : d(d)
    {
    }
    std::string col;
    COMPARE op;
    Data* d;
};
```

一个where子句查询包括了查询的列、操作符、比较值三个要素，

3.1.8 exception

所有的错误处理部分放在 API 层进行处理，API 层搜集下层的所有模块运行时产生的信息，以及自己预先处理数据的一些结果进行综合，通过 throw exception 的方式返回错误信息。而 Interepreter 再次将 exception 返回到 main 函数中，由 main 函数捕捉并输出错误信息。

3.2 Interpreter

Interpreter的类设计如下

```
class Interpreter
{
private:
    std::string query;

    std::vector<std::string> words;

    std::vector<int> isString;

    Api *api;

    void Pretreatment();

    void setWords();

    std::vector<WhereQuery> runWhere(int k);//这里的k意思是where后的第一个单词出现在
words的下标k处

    void runExecFile();

public:
    Interpreter();  // 因为 api 需要 new 出来，所以这里添加了构造函数
    ~Interpreter(); // 因为 api 需要 delete，所以加了析构函数

    void setQuery(std::string s);

    int runQuery();
};
```

运行的逻辑是，首先被main函数调用setQuery方法，设置一条新的SQL语句。对于分行的SQL语句，由main函数处理完毕，组合成一行的SQL语句后传入。

设置query为待执行的语句，我们首先对其进行预处理。预处理的内容包括，替换所有的制表符\t为空格，给所有单词两边加上空格，删除所有连续空格，删除开头和结尾的所有空格。

然后进行分词，也就是把SQL语句中的所有单词分开存储，这项工作可以极大地简化判断、数据转换等操作，大大提高运行效率，由于预处理时已经把所有单词按照空格分开，我们只需要以空格为界，存储单词进数组即可。

由于SQL语句的格式是固定的，我们只要根据相应位置的单词来判断语句的类型即可。例如，若第1个单词为select，则进入查询语句，则第2和第3个单词一定是*和from，否则报语法错；第4个单词是表名，查Catalog Manager，若无则报错；若有，则处理where子句；where子句每3个单词组成一个WhereQuery结构体，中间以and连接。整个处理过程只需要线性遍历1遍单词表即可。其他语句的处理方式类似。

3.3 API

API的类设计如下

```
class Api {
private:
    CatalogManager *cm;
    RecordManager *rm;
    IndexManager *im;
    BufferManager *bm;
public:
    Api();
    ~Api();

    void showTable(std::string tableName);
    void showTuple(Table *table);

    void createTable(std::string tableName, std::vector<Attribute> attr,
std::string primaryKey);
    void createIndex(std::string tableName, std::string indexName, std::string
colName);

    void dropTable(std::string tableName);
    void dropIndex(std::string indexName);

    void insertInto(std::string tableName, std::vector<Data*> data);
    Table* select(std::string tableName, std::vector<WhereQuery> wq);
    int deleteRecord(std::string tableName, std::vector<WhereQuery> wq);
};
```

- API类提供了所有待实现功能的接口，向上传递给解释器，包括：显示表、显示元组、创建表、创建索引、删除表、删除索引、插入、选择、删除元组。

  API类内置了下层的数个模块，即Catalog Manager，Buffer Manager，Index Manager和Record Manager。API模块接受解释器传来的离散的数据，将其整合入数据结构中，并且调用下层的模块的接口，再将结果返回回解释器。

  - **创建表**：直接调用 catalog manager 的创建表，判断是否合法，OK的话再把主键作为索引传给 index manager 创建索引文件。
  - **删除表**：直接调用 catalog manager 的删除表，判断是否合法，OK的话再把表的索引传给 index manager 删除索引文件。
  - **创建索引**：直接调用 catalog manager 的创建索引，判断是否有这个索引，有的话再给 index manager 创建索引文件。
  - **删除索引**：直接调用 catalog manager 的删除索引，判断是否有这个索引，有的话再给 index manager 删除索引文件。
  - **插入记录**：插入时首先判断该表是否有索引，如果有的话把利用索引来检查 unique 约束，如果没有则直接传给 record manager 处理 unique 约束和插入即可。最后得到插入的偏移地址，把他传给 index manager 进行相关索引的更新。
  - **删除记录**：删除时首先需要根据删除条件，利用 index manager 查询索引看是否能利用索引直接找到对应位置，如果可以则先将对应元组放在参数 Table 中，再传给 Record manager 进行操作，否则直接给 record manager 操作即可。
  - **查询记录**：查询时首先需要根据查询条件，利用 index manager 查询索引看是否能利用索引直接找到对应位置，如果可以则先将对应元组放在参数 Table 中，再传给 Record manager 进行操作，否则直接给 record manager 操作即可。

3.4 文件结构

上述的两个模块不需要和文件接触，从各个Manager开始，类将会接触到缓冲区以及文件，因此，我们需要先明确文件的存储结构。

3.4.1 inf文件

inf文件共有两个，分别是tbf.inf和idx.inf，顾名思义，分别存储所有的表名和所有的索引。都是以字符串存储的文本文件。

3.4.2 tbf文件

tbf文件存储Table的信息，格式如下

```
byte：属性数量          byte：索引数量    byte：主键位置
80个byte：属性名称      byte：类型       byte：长度（char类型）
……
80个byte：索引名词      byte：索引位置
……
```

开头3个字节存储属性数量、索引数量和主键位置，随后每个属性有82个字节来描述，每个索引有81个字节来描述。

一个文件最大是：$84 * 32 + 81 * 32 + 3 = 5283Byte$

3.4.3 rdf文件

rdf文件存储Table的所有记录信息，格式如下

```
byte: 是否被删除（0表示删除了，1表示没） n*byte:一条记录，与表的属性对应
byte: 是否被删除（0表示删除了，1表示没） n*byte:一条记录，与表的属性对应
byte: 是否被删除（0表示删除了，1表示没） n*byte:一条记录，与表的属性对应
......
```

注意，该文件不跨块存储。

3.5 Catalog Manager

CatalogManager类设计如下

```
class CatalogManager {
private:
    std::map<std::string, int> allTable;
    std::map<std::string, std::string> allIndex;

public:
    CatalogManager();
    ~CatalogManager();

    /* 新建表 */
    int createTable(Table &table);
    /* 删除表 */
    int dropTable(std::string tableName);
    /* 新建索引 */
    int createIndex(std::string tableName, std::string indexName, byte
indexNum);
```

```
    /* 删除索引 */
    int dropIndex(std::string indexName);
    /* 读取一张表的信息，如果不存在则返回 nullptr */
    Table* selectTable(std::string tableName);
    /* 展示表的信息 */
    void showTable(Table &table);
    /* 查询索引是否存在 */
    int queryIndex(std::string indexName);
};
```

每建立一个CatalogManager，都会把所有的表名和索引名载入内存；每删除一个CatalogManager，都会把所有的表名和索引名写入磁盘。这样保持了表信息的持久性。

- **新建表**：从自己的map变量中寻找是否已经存在，存在报错，不存在新建。
- **删除表**：从自己的map变量中寻找是否存在，存在则删除，不存在报错。
- **新建索引**：判断索引是否存在，存在则报错，不存在，把他保存到表信息中。
- **删除索引**：判断索引是否存在，不存在则报错，存在，把他删除。
- **读取表信息**：读取表文件，读取进来返回即可。
- **展示表**：按格式输出表信息即可。

3.6 Buffer Manager

Buffer Manager 类设计如下

```
const int Buffer_Number = 100;

class BufferManager {
private:
    Block buf[Buffer_Number];
    std::map<std::string, std::vector<Block*> > idx;
    std::list<Block*> pool;
    int time;

    Block* newBlock();  // 返回一个新的块，如果没有，LRU策略去掉一个块

public:
    BufferManager();
    ~BufferManager();

    Block* getBlock(std::string filename, int offset); // 返回该文件的 offset 块
    void writeBlock(Block* blk);  // 写标记
    void pinBlock(Block *blk);    // pin标记，锁住这个块
    void removeBlock(Block* blk); // 写回一个块到文件，强制执行

};
```

这里按新建到删除的过程介绍函数。

这里，一个块的大小为 4096 byte，为了提高运行速度，该模块使用一个链表管理内存池，每当一个缓冲块被请求时，该模块首先检查内存池是否存在这个缓冲块，如果存在则直接返回，如果不存在则从内存池中申请一段空间来读取文件进入缓冲块返回。如果内存池已经空了，那么该模块将会从已经存在的缓冲块中找到最早申请的那个缓冲块进行回收。

而当一个缓冲块被回收时，该模块根据是否有写标记来决定是否把内容写回文件中去，同时把缓冲块的内存还给内存池，提供下一次使用。

提供其他模块使用的就是 public 中的四个函数了，分别对应获取块，给块打上写记号（有写记号的块会在回收时写入文件中），锁住块（不可被自动回收），和强制回收块。

3.7 Index Manager

Index Manager 类设计如下

```
class IndexManager {
    BufferManager *bm;

    IndexManager(BufferManager *bm) : bm(bm) {};

    void createIndex(Table &t, int indexNo); //建立索引，参数：表（引用）与索引序号

    void deleteIndex(string indexName); //删除索引，参数：表（引用）与索引序号

    void insert(string indexName, Data* data, int offset); //插入记录，参数：索引名，数据与偏移量

    void eliminate(string indexName, Data* data); //删除记录，参数：索引名，数据

    int search(string indexName, Data* data); //查询记录，参数：索引名，数据

    std::vector<int> rangeSearch(string indexName, Data* inf, Data* sup); //范围查询记录，参数：索引名，数据下界，数据上界

};
```

Index Manager 需要使用 buffer manager，所以在类中需要传入进来。

- **创建索引**：创建索引只需要建立对应的.idx文件即可
- **删除索引**：删除索引只需要删除对应的.idx文件即可
- **插入**：插入一条新记录时，首先检查是否B+树中已经存在相关的键值，如果存在则返回错误信息。如果插入的新记录不包含重复的键值，则在B+树中插入这一键值，最后进行插入后的新树写入缓存区
- **删除**：删除记录时，直接自顶向底依层搜索节点，如果整棵树不存在相关键值，则返回错误信息。检查到有待删除键值的节点时，实时更新缓存块删除相关价值。这样以较高的效率实现了对B+树的更新。
- **查询**：查询和删除记录相仿，也是自顶向底依层搜索节点，如果不存在携带该键值的节点则返回-1，如果存在这样的节点则返回偏移量
- **范围查询**：范围查询获得在一定区间范围内一系列节点的偏移量集合。其过程想到与对下界先进行一次普通的查询，然后再遍历起始节点之后的后继叶节点，直到键值遍历到上界或遍历完所有叶节点。

3.8 Record Manager

Records Manager 类设计如下

```
class RecordManager {
private:
    BufferManager *bufferManager;
public:

    RecordManager(BufferManager *bm);
```

```
    /* 创建表与删除表 */
    void createTable(std::string tableName);
    int dropTabel(std::string tableName);
    /* insert 语句，新加一个元组 */
    int insert(Table *table, Tuple tuple);
    /* delete 语句，被删除的记录会存在 table 的元组中 */
    Table* del(Table *table, std::vector<WhereQuery> &wq);
    /* select 结果会保存在 tabel 中 */
    int select(Table *table, std::vector<WhereQuery> &wq);


    // 都是内部使用的函数，但是为了方便，也直接放在public中
    void addRecord(Table *table, int address);
    void deleteRecord(Table *table, int address);
    bool judge(Table *table, std::vector<WhereQuery> &wq, Tuple *tuple);
    Tuple* readRecord(Table *table, Block *blk, int pos);
    void choose(Table *table, std::vector<WhereQuery> &wq);
    int calcLength(Table *table);
};
```

Record Manager 需要使用 buffer manager，所以在类中需要传入进来。

- **创建与删除**：创建表与删除表只需要建立或删除对应的文件即可，不需要进行其他额外的操作。
- **插入**：插入一条新记录时，需要对表属性的 unique 约束进行检查，也就是遍历表格所有记录，进行约束检查，约束正确即寻找到第一个可行的插入位置（也就是一条已经被删除的记录，或者是新的空间）。在该位置进行插入之后，返回插入的位置。（API 层可能会调用 index manager 插入索引记录）
- **删除**：删除记录时，首先检查自己传进来的 table 内是否已经存在元组，如果存在则说明 API 层调用了 index manager 已经得到了一些数据，那么删除时直接根据结果进行删除即可。而如果没有，就说明删除条件中并不存在可以索引的属性，那么就只能遍历所有记录，进行删除了。
- **查询**：查询和删除记录类似，首先检查自己传进来的 table 内是否已经存在元组，如果存在则说明 API 层调用了 index manager 已经得到了一些数据，那么查询只需要再做进一步筛选即可（因为可能条件中有的属性没有索引，需要在结果中进一步筛选）。而如果没有，就说明查询条件中并不存在可以索引的属性，那么就只能遍历所有记录，进行查询了。

4 测试

4.1 正确性测试

这里，我们设计了一个规模小的测试文件进行测试，其中有正确和错误的指令，来测试程序运行的正确性。

以下是文件内容：

```
create table test(
    a int,
    b float,
    c char(10),
    primary key(a)
);

insert into test values(1, 2, '12');
insert into test values(2, 3, '13');
insert into test values(3, 4, '15');
insert into test values(2, 5, '13');
```

```
select * from test where b<>4;
select * from test where a>=2;

delete from test where a=3;
insert into test values(3, 8, 'aaaaaa');
insert into test values(2, 3, '13');

create table test(
    a int,
    b int unique,
    c float,
    primary key(a)
);

drop table test;

create table test(
    a int,
    b int unique
    primary key(a)
);

insert into test values (1,2);
insert into test values (2,3);
insert into test values (3,4);
insert into test values (4,5);

insert into test values (5,2);
insert into test values (5,6);

select * from test;

create index bi on test (b);
select * from test where b<=4;

drop index bi;
```

下面是程序输出：

```
MiniSQL: create table successfully!
---test---
a int unique primary key
b float
c char(10)
index: a_test(a)
MiniSQL: insert successfully!
MiniSQL: insert successfully!
MiniSQL: insert successfully!
MiniSQL: error: insert failed beacuse of the unique attribute a
===== test ======
a       b       c
1       2       12
2       3       13
===== test ======
a       b       c
2       3       13
3       4       15
```

```
MiniSQL: delete 1 records!
MiniSQL: insert successfully!
MiniSQL: error: insert failed beacuse of the unique attribute a
MiniSQL: error: table already exists!
MiniSQL: drop table successfully!
MiniSQL: create table successfully!
---test---
a int unique primary key
b int unique
index: a_test(a)
MiniSQL: insert successfully!
MiniSQL: insert successfully!
MiniSQL: insert successfully!
MiniSQL: insert successfully!
MiniSQL: error: insert failed beacuse of the unique attribute b
MiniSQL: insert successfully!
===== test ======
a       b
1       2
2       3
3       4
4       5
5       6
MiniSQL: create index successfully!
---test---
a int unique primary key
b int unique
index: a_test(a)
index: bi(b)
===== test ======
a       b
1       2
2       3
3       4
MiniSQL: drop index successfully!
```

可以看到，所有指令运行正常。

4.2 性能测试

我们设计了一组 2000 次插入的大数据，其大致如下：

```
create table test(a int, b char(255) unique, primary key(a));
insert into test values (0, '很长的字符串');
insert into test values (1, '很长的字符串');
.....
insert into test values (1999, '很长的字符串');
```

每次都要进行 255 个字符的 unique 约束检查，其运行结果如下。

```
.....
MiniSQL: insert successfully!
MiniSQL: insert successfully!
MiniSQL: insert successfully!
8.98 seconds
```

一共使用了 8.98 s，可以看出在适中数据量下，速度较快。

我们设计了一组 4000 次插入的大数据，其大致如下：

```
create table test(a int, b char(255) unique, primary key(a));
insert into test values (0, '很长的字符串');
insert into test values (1, '很长的字符串');
.....
insert into test values (3999, '很长的字符串');
```

每次都要进行 255 个字符的 unique 约束检查，其运行结果如下。

```
.....
MiniSQL: insert successfully!
MiniSQL: insert successfully!
MiniSQL: insert successfully!
55.87 seconds
```

一共使用了 55.87 s，速度可以接受。