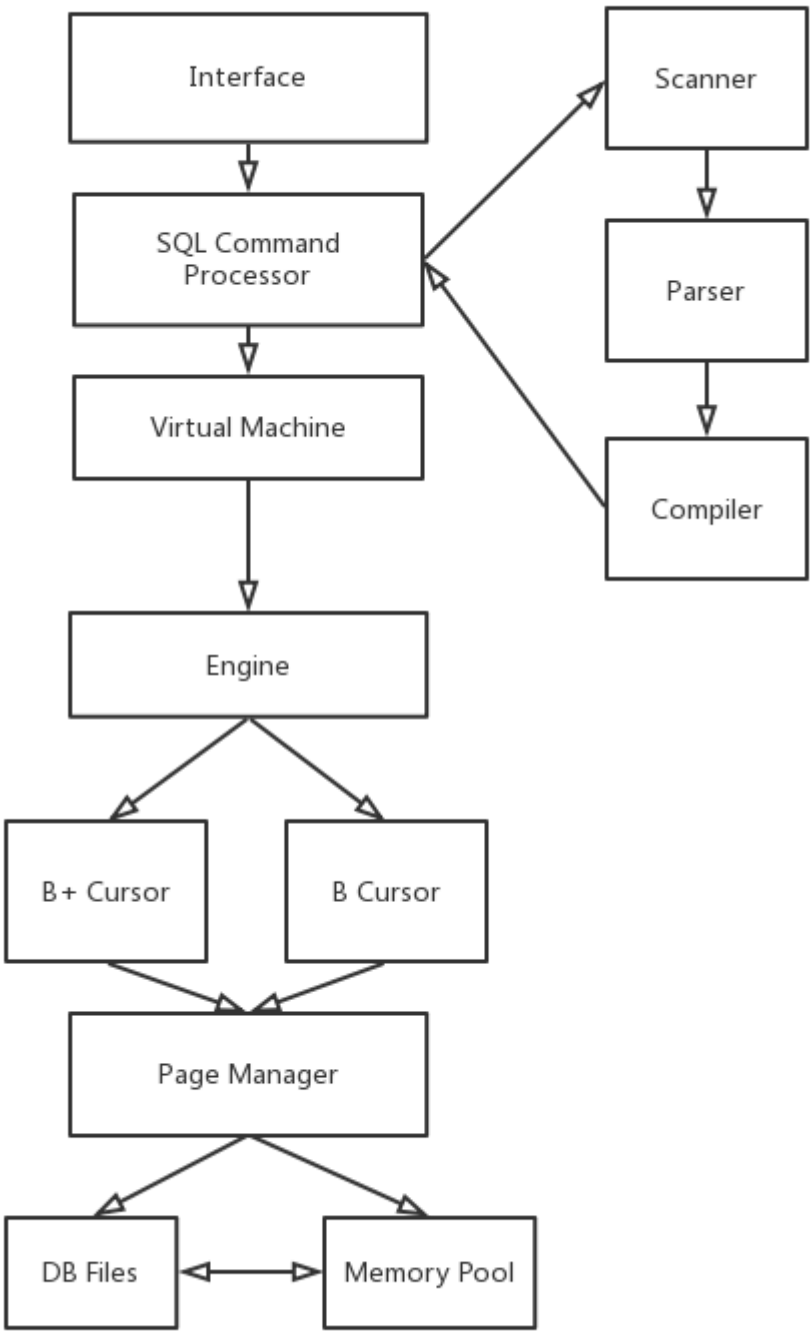
1. MiniSQL总体框架

1.1 MiniSQL 实现功能描述

- 总功能：允许用户通过字符界面输入SQL语句实现表的建立/删除；索引的建立/删除以及表记录的插入/删除/查找。
- 数据类型：支持三种基本数据类型：int，char(n)，float，其中char(n)满足$1 \leq n \leq 255$。
- 表定义：一个表最多可以定义32个属性，各属性可以指定是否为unique；支持单属性的主键定义。
- 索引的建立和删除：对于表的主属性自动建立B+树索引，对于声明为unique的属性可以通过SQL语句由用户指定建立/删除B+树索引（因此，所有的B+树索引都是单属性单值的）。
- 查找记录：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。
- 语法说明：MiniSQL支持标准的SQL语句格式，每一条SQL语句以分号结尾，一条SQL语句可写在一行或多行，所有的关键字都为小写。具体语法请参照标准SQL的语法。

1.2 MiniSQL系统设计结构

软件体系结构图



功能模块概述

Parser 模块

Parser的主要功能为读入字符流后格式化为可用于内部交流的数据结构，并在此阶段初步过滤基本SQL语法中的错误。

Compiler 模块

Compiler负责将Parser处理后的结构化信息进一步编译为可在后端引擎运行的字节码，并打包必要数据、提供运行时环境。 为了使用方便，这一模块将直接提供与用户交互的接口进行数据库操作。

Engine 模块

Engine模块承接上一层的调用请求，提供了很多易于组合的接口。这一模块同时提供信息接口，承担部分Catalog功能，支持Compiler模块根据数据库情况进行具体编译。

Cursor 类

Engine模块只是提供了操纵接口，在更底层，这些操作都由具体的Cursor实现。在本项目中实现了BFlow Cursor用于读写数据库表记录，BPlus Cursor用于读写数据库索引记录。

PageManager 模块

所有针对文件及分页进行的读写操作全部由PageManager进行接管，PageManager在页为单位的抽象层实现了基本的并发控制，控制多线程访问的权限及底层事务。

其他功能性类

本工程使用了多种C++编程范式来简化程序设计的逻辑，其中反射功能为多种类型数据的多种转换提供了运行时支持，基于可变参模板实现的强功能字典为上下文无关文法的设计和运行提供了直观方法，其他一些小型功能模块比如继承于levelDB的Status信息类、自定义扩展的堆栈和哈希表类、二进制封装Blob类、基于C++接口的并行锁类，都为各类高级功能提供了支持。

1.3 运行环境和参考配置

本MiniSQL项目使用纯C++开发，并在其中使用了C++正则匹配库，以及C++11特性如函数绑定、原子变量等，因此需要支持C++11以上（包含<regex>头文件）的编译器进行编译。

1.4 参考资料

- 'SQLite' online document
- 'LevelDB' source code
- 'Mushroom' source code
- 《Transaction Processing: concept and techniques》. by Jim Gray & Andreas Reuter

2. MiniSQL各模块设计介绍

2.1 Parser 模块

本模块源文件包含于`Compiler`目录下的`parser.hpp`和`scanner.hpp`中。

Parser功能主要被分为两个阶段：

第一个分析流程为词法分析，基于简单的SQL支持要求，本工程中编码了34种基本词，并使用`regex`库中的正则引擎进行分析。具体的词法如下：

```
static const AutoDict<std::string> kWordDict(
  "NUMBER","STRING",
  "SEMICOLON","COMMA","LBRACKET","RBRACKET","DOT",
  "SELECT","FROM", "WHERE",
  "INSERT","INTO","VALUES",
  "CREATE","TABLE","PRIMARY","KEY","CHAR","INT","FLOAT","UNIQUE",
  "DELETE",
  "DROP","INDEX","ON",
  "UNARY_LOGIC_OP","LOGIC_OP","BOOL_OP",

  "TERM_OP","MULTIPLY_OP","FACTOR_BINARY_OP","FACTOR_UNARY_OP",
```

```
  "NAME"
  );
static const char* kWordLex[kWordSize] = {
  "^[\r\t\n ]",  // none
  "^([0-9]+|[0-9]+[.][0-9]*)",
  "^\'[^\']+\'",
  "^;",
  "^,",
  "^[(]",
  "^[)]",
  "^[.]",
  "^(select|SELECT)[\r\t\n ]",
  "^(from|FROM)[\r\t\n ]",
  "^(where|WHERE)[\r\t\n ]",
  "^(insert|INSERT)[\r\t\n ]",
  "^(into|INTO)[\r\t\n ]",
  "^(values|VALUES)",
  "^(CREATE|create)[\r\t\n ]",
  "^(TABLE|table)[\r\t\n ]",
  "^(PRIMARY|primary)[\r\t\n ]",
  "^(KEY|key)",
  "^(CHAR|char)",
  "^(INT|int)",
  "^(FLOAT|float)",
  "^(UNIQUE|unique)",
  "^(DELETE|delete)",
  "^(DROP|drop)[\r\t\n ]",
  "^(INDEX|index)[\r\t\n ]",
  "^(ON|on)[\r\t\n ]",
  "^(not|NOT)[\r\t\n ]",
  "^(and|AND|or|OR)[\r\t\n ]",
  "^(<|<=|>|>=|=|!=|like|LIKE|in|IN)",
  "^[+-]",
  "^[*]",
  "^[/]",
  "^(sqrt)",
  "^[a-zA-Z_][a-zA-Z0-9_]*"
    };
```

第二个分析流程为语法分析，也是整个过程中复杂度最高的部分。具体过程为Parser模块使用预测方法根据预先设计的上下文无关文法产生式进行广度优先分析，在符号比对的过程中淘汰不合理推测。

基于简单的SQL类型，本工程中使用了60条产生式进行语法推理，具体如下。

```
static LayeredDict<int, Stack<int>> kRuleDict(
  _from_("sql"),_to_("select_clause","SEMICOLON"),
  _from_("sql"),_to_("insert_clause","SEMICOLON"),
  _from_("sql"),_to_("create_clause","SEMICOLON"),
  _from_("sql"),_to_("delete_clause","SEMICOLON"),
  _from_("sql"),_to_("drop_clause","SEMICOLON"),
  _from_("sql"),_to_("create_index_clause","SEMICOLON"),
  _from_("select_clause"),_to_("SELECT", "column_list","FROM","table_list","where_clause"),
  _from_("insert_clause"),_to_("INSERT","INTO","table_list","package_list"),
```

```
  _from_("delete_clause"),_to_("DELETE","FROM","table_list","where_clause"),
  _from_("drop_clause"),_to_("DROP","INDEX","NAME"),
  _from_("drop_clause"),_to_("DROP","TABLE","NAME"),
  _from_("create_index_clause"),_to_("CREATE","INDEX","NAME","ON","NAME","LBRACKET","NAME","RBRA
CKET"),
  _from_("package_list"),_to_("package","package_list_tail"),
  _from_("package_list_tail"),_to_("COMMA", "package","package_list_tail"),
  _from_("package_list_tail"),_to_("NONE"),
  _from_("package"),_to_("VALUES","LBRACKET","value_list","RBRACKET"),
  _from_("value_list"),_to_("value_expr","value_list_tail"),
  _from_("value_list_tail"),_to_("COMMA","value_expr","value_list_tail"),
  _from_("value_list_tail"),_to_("NONE"),
  _from_("create_clause"),_to_("CREATE","TABLE","NAME","LBRACKET","field_list","RBRACKET","NONE"
),
  _from_("field_list"),_to_("NAME","type","field_list_tail"),
  _from_("field_list"),_to_("NAME","type","UNIQUE","field_list_tail"),
  _from_("field_list_tail"),_to_("COMMA","NAME","type","field_list_tail"),
  _from_("field_list_tail"),_to_("COMMA","NAME","type","UNIQUE","field_list_tail"),
  _from_("field_list_tail"),_to_("COMMA","PRIMARY","KEY","LBRACKET","column_list","RBRACKET","fi
eld_list_tail"),
  _from_("field_list_tail"),_to_("NONE"),
  _from_("type"),_to_("INT"),
  _from_("type"),_to_("FLOAT"),
  _from_("type"),_to_("CHAR","LBRACKET","NUMBER","RBRACKET"),
  _from_("column_list"),_to_("MULTIPLY_OP","NONE"),
  _from_("column_list"),_to_("column","column_list_tail"),
  _from_("column"),_to_("NAME", "DOT", "NAME"),
  _from_("column"),_to_("NAME", "DOT", "MULTIPLY_OP"),
  _from_("column"),_to_("NAME"),
  _from_("column_list_tail"),_to_("NONE"),
  _from_("column_list_tail"),_to_("COMMA","column","column_list_tail"),
  _from_("table_list"),_to_("NAME", "table_list_tail"),
  _from_("table_list_tail"),_to_("NONE"),
  _from_("table_list_tail"),_to_("COMMA","NAME","table_list_tail"),
  _from_("where_clause"),_to_("NONE"),
  _from_("where_clause"),_to_("WHERE","condition_clause"),
  _from_("condition_clause"),_to_("single_condition","condition_tail"),
  _from_("condition_tail"),_to_("NONE"),
  _from_("condition_tail"),_to_("LOGIC_OP","single_condition","condition_tail"),
  _from_("single_condition"),_to_("UNARY_LOGIC_OP","single_condition"),
  _from_("single_condition"),_to_("LBRACKET","condition_clause","RBRACKET"),
  _from_("single_condition"),_to_("value_expr","BOOL_OP","value_expr"),
  _from_("value_expr"),_to_("term","term_tail"),
  _from_("value_expr"),_to_("select_clause"),
  _from_("term_tail"),_to_("TERM_OP","term","term_tail"),
  _from_("term_tail"),_to_("NONE"),
  _from_("term"),_to_("factor","factor_tail"),
  _from_("factor_tail"),_to_("NONE"),
  _from_("factor_tail"),_to_("FACTOR_BINARY_OP","factor","factor_tail"),
  _from_("factor_tail"),_to_("MULTIPLY_OP","factor","factor_tail"),
  _from_("factor"),_to_("LBRACKET","value_expr","RBRACKET"),
  _from_("factor"),_to_("FACTOR_UNARY_OP","factor"),

  _from_("factor"),_to_("column"),
```

```
  _from_("factor"),_to_("NUMBER"),
  _from_("factor"),_to_("STRING")
  );
```

一些比较简单的元语句比如execfile以及quit没有被编码入这套语法分析体系中。

而在这一推理过程的具体实现中，我没有完全使用语法树形式存储结构化信息，而是使用序列化的符号流。符号流中包含着已推理符号，每个已推理符号中携带了原文字串和推理规则编号。通过这种简单的装饰也能基本表达整个语法树的信息内容，并在Compile阶段恢复和重整。

2.3 Compiler 模块

Compiler模块实际上承担了多种功能，具体来说有以下几个：分析编译推理符号流；运行编译产物；提供交互界面。

首先是最简单的交互接口部分，Compiler模块提供以下函数：

```
void RunInterface(istream& is, ostream& os, bool prompt = true);
```

基于输入输出流，Compiler能建立固定的交互界面。在CLI上，它模仿MySQL的命令行界面，在文件流上，它提供控制变量来控制是否输出提示词。交互程序的主循环伪码如下：

```
RunInterface(istream, ostream):
    while istream not end:
        ostream << prompt
        if 'execfile' in istream:
            RunInterface(file.istream, ostream)
        Scanner.Scan(istream)
        Compile()
        RunInstructions(ostream)
        Clear()
```

其次是编译功能部分，通过对固定语法结构的识别，在Compile阶段可以将语法流分派为几种类型分别编译，分别是`CompileSelect`，`CompileInsert`，`CompileCreate`，`CompileDelete`。在子流程中，程序将语法流中的符号进行提取分析，并将字符形式的数值进行反序列化和打包。具体来说，不同的类型编译共用几个元编译过程：`ParseSimpleExpr`用于分析计算字面量数值并打包为字符串，`ParseCondition`用于分析多个AND连接的条件语句。

这一系列的编译方法的最终产物为指令码序列和资源集合，下面简单介绍本工程中使用到的指令码及各自的资源需求：

```
enum ByteCode{
    kTransaction,
    kOpenRecordCursor,
    kOpenIndexCursor,
    kPrepareMatch,
    kPrepareSequence,
    kNextSlice,
    kIf,

    kIfNot,
```

```
    kIfNil,
    kJump,
    kInsertSlice,
    kDeleteSlice,
    kPrintSlice
};
```

如上所示，为了实现基本SQL命令所需要的指令码是很少的，其中一部分指令如`kIf`,`kIfNot`,`kIfNil`,`kJump`为跳转指令，它们需要携带指令跳转Offset作为指令数据，同时，它们的跳转条件也需要不同的数据支持：`kIf`需要将全局数据通过全局条件过滤器进行检查、`kIfNil`则测试全局数据的可用性。

另一部分指令直接转化为数据库引擎操作，`kOpenIndexCursor`命令打开引擎中的特定索引，需要提供域名称作为参数；`kPrepareMatch`命令向数据库提交读取请求，并准备相等数据，需要提供一个match值作为参数；`kPrepareSequence`类似，需要提供min/max两个范围值作为参数；`kNextSlice`，`kInsertSlice`，`kDeleteSlice`，`kPrintSlice`系列指令共用一个全局资源，也即一个记录指针，它们基于这一共用指针进行合作并实现更复杂的操作。

下面使用一个简单的select语句来分析整个流程：

- 分析符号：分析语法流中指定的select域名称和目标表名称，并打包至全局资源中的`DisplayColumns`和`Tables`中

  这一步后编译器生成以下语句：

  ```
  kTransaction
  kOpenRecordCursor  (使用全局Table变量)
  ```

- 分析条件：根据语法流迭代分析不同AND字句。

  对于每一条AND字句，根据数据库索引情况计算其Rank值并更新最佳索引查找选择，Rank值的排序大致如下：主键Match > 附键Match > 主键 Sequence > 附键Match > 主键全局搜索

  同时，将AND的具体内容编码存贮，这里使用C++11提供的函数式范式，将每一字句内容编码为Callable对象`std::function<bool(const Slice&)>`，存入全局资源

  这一步后编译器生成以下语句

  ```
  kOpenIndexCursor FieldName  (使用全局Table变量)
  kPrepareMatch Match 或 kPrepareSequence Min,Max
  ```

- 完成编码：在以上语句之后追加完成Select功能

  ```
  .NEXT:
  kNextSlice
  kIfNil #END
  kIfNot #NEXT
  kPrintSlice
  kJump #NEXT
  .END:
  ```

最后，Compiler还提供运行时环境来执行字节码指令：使用pc指针遍历字节码序列，使用switch-case结构分别执行语句。

2.4 Engine 模块

主要功能

1. 对接Compiler指令码功能，提供基本元指令如建表和索引操作、运行时数据句柄操作如OpenCursor/CloseDatabase、数据读写如NextSlice
2. 提供数据库元信息的查询结构，比如GetSchema、CheckDatabase等

接口设计

外部接口

```
// 元命令
Status CreateDatabase(std::string path = kDatabaseRootPath);
Status CreateTable(Schema& schema);
Status LoadDatabase(std::string path = kDatabaseRootPath);
Status LoadTable(std::string name);
Status DropDatabase(void);
Status DropTable(std::string name);
Status DropIndex(std::string IndexName);
Status CloseDatabase(void);
Status MakeIndex(std::string table, std::string field, std::string name);
// 事务操作
Status Transaction(void);
Status OpenCursor(std::string table);
Status OpenCursor(std::string table, std::string field);
Status PrepareMatch(Value* match);
Status kPrepareSequence(Value* min, Value* max, bool leftEqual, bool rightEqual);
// 数据操作
Status NextSlice(SlicePtr& ret);
Status InsertSlice(Slice* slice);
Status DeleteSlice(Slice* slice);
```

为了最大程度简化外部调用，这里的数据操作使用了比较有趣的接口设计NextSlice，对于典型的数据库查询，这一接口的实现逻辑为：

- 事务准备阶段：Prepare接口将准备PageHandle并存储在Engine内部数据中，对于主键查询，只需要存储第一页句柄
- 数据查询阶段：若Engine内部没有缓存记录，则调去内部的Handle缓存，对此Handle进行数据缓存，取缓存中的第一个返回。
- 数据修改阶段：此时，Engine内部的第一个Handle缓存即为刚才数据查询的文件句柄，因此以此Handle作为修改句柄，执行具体修改请求。

2.5 Cursor 类

主要功能

Cursor为Engine的具体操作提供了一个便捷的实施句柄，Cursor与底层的PageManager相连，能够直接读写具体分页的信息，同时Cursor内保存状态，可以实现多线程下较为细粒度的数据操作。

接口设计

首先来看不同Cursor类内的数据成员：

```
// BFlowCusror 数据成员 //
// maintenence
PageManager* page_;
PageHandle root_;
Schema* schema_;
// runtime
struct BFlowPageInfo{
  FileHandle hFile;
  PageHandle hPage;
  PageHandle hPri;
  PageHandle hNext;
  uint16_t nSize;
} set_;

// BPlusCursor 数据成员 //
// maintenance
PageManager* page_;
int key_len_;
Schema* schema_;
PageHandle root_;
struct BPlusPageInfo{
  FileHandle hFile;
  // history handles
  PageHandle hTrace[kBPlusStackSize];
  int nStackTop; // stack helper
  PageHandle hPage;
  uint16_t nSize;
  PageHandle hRight;
  PageHandle hDown;
} set_;
```

可以看到，每个Cursor类内有两部分数据，一是关于此Cursor附着索引或表的数据信息，同时含有操纵文件的Manager引用；二是运行时状态数据，包含了关于当前指向数据分页的必要数据，在BPlusCursor索引光标类内，还有堆栈形式保存的按高度排序的历史分页记录，尽管这一设计在多线程运行下有一定概率失去一致性，但对B-link具体算法的微调应该可以解决这一问题。

除此之外，每种Cursor提供了相似的操作接口：

```
// Initialize //
inline void Set(Schema* schema, PageHandle root);
// Movement //
inline Status ShiftRight(void);
inline Status ShiftLeft(void);
inline Status Rewind(void);
// Accessor //
inline PageHandle rightHandle(void);
inline PageHandle leftHandle(void);
inline size_t size(void);
inline PageHandle currentHandle(void);
```

```
// IO //
Status Get(Value* min, Value* max, bool& left, bool& right, SliceContainer& ret);
Status InsertOnSplit(Slice* slice, PageHandle& ret);
Status Delete(Slice* record);
Status Insert(Slice* record);
```

这些操作提供了完全封闭于当前分页的操作，提高细粒度，同时减少了并发控制的复杂度。

2.6 Page Manager 模块

主要功能

1. 封装的文件操作：文件及分页的创建和删除
2. 带并发控制的分页读写接口
3. 文件元数据快速查询接口

接口设计

Page Manager类包含所有以页为抽象单位的读写操作，分为文件操作、分页操作、同步操作、以及信息查询，它们的具体实现接口如下：

```
// 文件操作
Status NewFile(string FileName, uint8_t BlockSize, FileHandle& ret);
Status OpenFile(string FileName, FileHandle& ret);
Status CloseFile(FileHandle hFile);
Status DeleteFile(FileHandle hFile);
// 分页操作
Status NewPage(FileHandle hFile, PageType type, PageHandle& ret);
Status DeletePage(PageHandle hPage);
// 分页同步操作
Status SyncFronFile(PageHandle hPage);
Status SyncFromMem(PageHandle hPage);
Status DirectWrite(PageHandle hPage, void* ptr);
Status Pool(PageHandle hPage);
Status Expire(PageHandle hPage);
// 信息查询
size_t GetFileSize(FileHandle hFile);
size_t GetPageSize(PageHandle hPage);
bool fileIsOpened(FileHandle hFile);
PageType GetPageType(PageHandle hPage);
```

模块内部设计

PageManager内存储了两个数据类：file_ 和 pool_

其中**file**_为哈希表实现的FileHandle至FileWrapper映射，**pool**_为MemPool类实现的PageHandle至内存指针映射。

大部分与文件和分页相关的数据都存贮在FileWrapper中，这一数据类包含大量信息，列举如下：

```
// struct FileWrapper
FilePtr file; // 实际文件类，实现了所有操作系统接口的封装方法
```

```
// 额外数据
int BlockSize;
FileHandle hFile;
string FileName;
size_t DataOffset;
// 分页数据
std::vector<PagePtr> pages;
// 文件级别同步
Latch latch;
atomic<size_t> page_size;
// FreeList
atomic<size_t> free_index;
atomic<size_t> free_size;
```

在文件级别，使用单个锁进行大规模修改的并发控制，同时使用Page类包装分页信息，每个文件包含的分页句柄是简单的FileHandle | PageNo

以下为分页级别数据类成员：

```
PageHandle handle;
Latch latch;
PageType type;
TimeType modified;
TimeType commited;
```

分页类中的并发控制锁是运行时最常使用的控制句柄，用于控制任何读写操作。

分页类中包含了修改和提交的时间戳记录域，主要用于减少不必要的磁盘IO操作。

值得注意的是，在PageManager公开接口中，没有提供直接访问数据的方法。这是因为为了严格控制并发访问，我在项目中使用了RAII风格的访问方法，具体由PageRef实现：

```
struct PageRef: public NoCopy{
  PageManager* manager; // global
  PageHandle handle;
  DeducedRuntimeAccessMode mode;
 public:
  char* ptr;
  PageRef(PageManager* manage, PageHandle page, RuntimeAccessMode rmode);
};
```

要访问数据，必须要通过PageManager应用和分页句柄以及访问模式建立访问对象，在PageRef的内部会根据提交的访问类型进行适当的并发锁操作，并在对象析构时自动释放资源。

本工程中使用了以下几种访问请求:

```
enum RuntimeAccessMode{
  kEmptyAccess = 0,
  kReadOnly = 1,
  kLazyModify = 2,
  kFatalModify = 3,
  kIncrementalModify = 4
};
```

其中LazyModify指内存修改，FatalModify指直接磁盘读写修改，IncrementalModify指不排读型修改。只要访问者遵守其请求的访问模式，整个系统的一致性应当能得到保证。

2.7 其他重要模块

2.7.1 Latch 类

基于C++11的mutex/condition_variable/atomic库，我实现了比较基本的高级读写锁，包含三种读写类型，并提供访问升级操作。

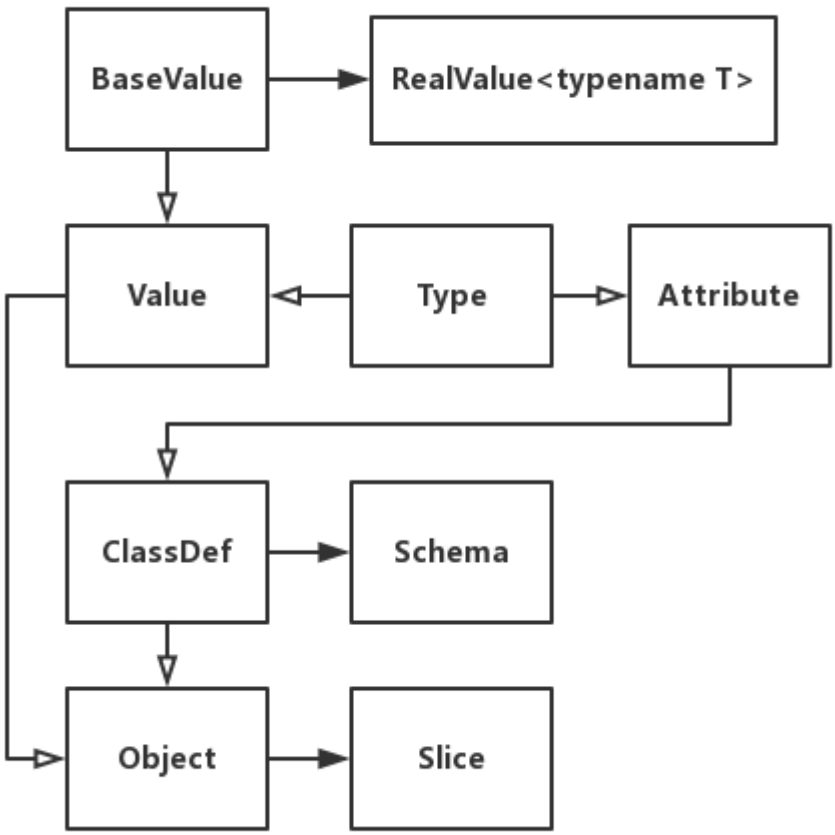
```
class Latch: public NoCopy{
    std::mutex mutex_;
    // reader
    std::atomic<size_t> readers_;
    std::condition_variable cond_r_;
    // writer
    std::atomic<size_t> strong_write_appliers_; // queue
    std::atomic<size_t> weak_write_appliers_;
    bool swriter_;
    std::condition_variable cond_sw_;
    bool writer_; // writer = strong + weak
    std::condition_variable cond_ww_;
public:
    Latch(): readers_(0),
        weak_write_appliers_(0),strong_write_appliers_(0),
        writer_(false),swriter_(false){ }
    ~Latch(){ }
    void ReadLock(void){
        // cannot acquire when strong writer is applying
        // strong writer is banned
        std::unique_lock<std::mutex> local(mutex_);
        cond_r_.wait( local, [=]()->bool {return !swriter_;} );
        readers_ ++;
    }
    void WeakWriteLock(void){ // writer is banned
        std::unique_lock<std::mutex> local(mutex_);
        weak_write_appliers_ ++;
        cond_ww_.wait( local, [=]()->bool{ return !writer_; } );
        writer_ = true;
        weak_write_appliers_ --;
    }
    void WriteLock(void){ // writer and reader are banned

        std::unique_lock<std::mutex> local(mutex_);
```

```
        strong_write_appliers_ ++;
        cond_sw_.wait(local, [=]()->bool{return readers_ == 0 && !writer_;});
        swriter_ = true;
        writer_ = true;
        strong_write_appliers_ --;
    }
    void ReleaseReadLock(void){ // notify strong writer
        std::unique_lock<std::mutex> local(mutex_);
        if(--readers_ >= 1){
            // assert no writers
            if(strong_write_appliers_ > 0){
                cond_sw_.notify_one();
            }
        }
    }
    void ReleaseWeakWriteLock(void){ // notify weak writer
        std::unique_lock<std::mutex> local(mutex_);
        writer_ = false;
        // assert no strong writer
        if(weak_write_appliers_ > 0){
            cond_ww_.notify_one();
        }
        else if(readers_ >= 1 && strong_write_appliers_ > 0){
            cond_sw_.notify_one();
        }
    }
    void ReleaseWriteLock(void){ // notify weak writer -> writer -> all reader
        std::unique_lock<std::mutex> local(mutex_);
        writer_ = false;
        swriter_ = false;
        if(weak_write_appliers_ > 0){
            cond_ww_.notify_one();
            cond_r_.notify_all();
        }
        else if(strong_write_appliers_ > 0){
            cond_sw_.notify_one();
        }
        else{
            cond_r_.notify_all();
        }
    }
    void ReadLockLiftToWriteLock(void){
        // assert having acquire a read lock
        std::unique_lock<std::mutex> local(mutex_);
        strong_write_appliers_++;
        cond_sw_.wait(local, [=]()->bool{return readers_ == 1;});
        readers_--; // now release read lock
        swriter_ = true;
        writer_ = true;
        strong_write_appliers_ --;
    }
    void WeakWriteLockLiftToWriteLock(void){

        std::unique_lock<std::mutex> local(mutex_);
```

```
        strong_write_appliers_ ++;
        cond_sw_.wait(local, [=]()->bool{return readers_ == 0;}); // already hold weak write
lock
        swriter_ = true;
        strong_write_appliers_ --;
    }
    bool occupied(void){
        return readers_ > 0 || writer_;
    }
}; // class Latch
```

2.7.2 Reflection 实现

为了更方便统一地操纵多种运行时类型数据，我实现了简单的多层封装的反射类库，其中包含了一些类型如`Value`、`BaseValue`、`RealValue`、`Attribute`、`ClassDef`、`Object`，它们的依赖关系如下图：



图中实心箭头表示类型继承，空心箭头表示成员包含关系。

其中BaseValue作为纯虚基类作为不同类型的引用句柄方便反射系统内部的互相沟通；RealValue模板类继承自BaseValue，实际拥有基本类型数据；最终Value类封装所有RealValue实现的方法，承担起公共接口的角色。

除了简单的C++基本数据类型，这套类型系统支持任何自定义数据类型，对数据类型的要求有以下几点：

- 提供对std::string的转换接口
- 提供对C++ iostream输入输出流的友元函数
- 提供二进制数据的指针接口

- 完整的类型数值比较方法

在基础的类型系统之上，模拟C++类的反射机制得以建立。Attribute模拟C++成员符号，包含了命名的字符串表达和类型元信息；ClassDef模拟C++类型定义，包含顺序成员符号，并支持简单继承关系；最后，Object模拟C++成员实例，在保留对ClassDef引用基础上包含了数据成员的实际值。

经过进一步的特别包装，ClassDef和Object类型就可以作为数据库系统中的表定义数据Schema和表记录数据Slice使用。

3. 测试方案和测试样例

3.1 模块测试

Scanner模块

Scanner调用C++11正则引擎，实现简单的识别功能，在实际操作中可以通过命令行交互界面实时测试，具体例子如下：

```
>> select * from test;
SELECT, MULTIPLY_OP, FROM, NAME, SEMICOLON
>> create table test where a != 0;
CREATE, TABLE, NAME, WHERE, NAME, BOOL_OP, NUMBER, SEMICOLON
>> drop index on test;
DROP, INDEX, ON, NAME, SEMICOLON
```

注意到这一阶段，Scanner模块不具备对语法进行分析查错的功能，对各种异常语句进行正常解析。

Parser 模块

Parser模块使用预定义上下文无关文法来进行语法解析，在模块测试中主要采用正确样例的结果比照方法，对需要支持的几类SQL语句进行交互式分析，并验证语法结构正确性。

一些为典型SQL语句的语法分析结果：

```
>> select * from test where a != 9 and b > 0 ;
sql(0), select_clause(0), SELECT(-1)[select ], column_list(0), MULTIPLY_OP(-1)[*], NONE(-1)[from
], FROM(-1)[from ], table_list(0), NAME(-1)[test], table_list_tail(0), NONE(-1)[where ],
where_clause(1), WHERE(-1)[where ], condition_clause(0), single_condition(2), value_expr(0),
term(0), factor(2), column(2), NAME(-1)[a], factor_tail(0), NONE(-1)[!=], term_tail(1), NONE(-1)
[!=], BOOL_OP(-1)[!=], value_expr(0), term(0), factor(3), NUMBER(-1)[9], factor_tail(0),
NONE(-1)[and ], term_tail(1), NONE(-1)[and ], condition_tail(1), LOGIC_OP(-1)[and ],
single_condition(2), value_expr(0), term(0), factor(2), column(2), NAME(-1)[b], factor_tail(0),
NONE(-1)[>], term_tail(1), NONE(-1)[>], BOOL_OP(-1)[>], value_expr(0), term(0), factor(3),
NUMBER(-1)[0], factor_tail(0), NONE(-1)[;], term_tail(1), NONE(-1)[;], condition_tail(0),
NONE(-1)[;], SEMICOLON(-1)[;] $$

>> create table test (sno int unique, sname char(9), primary key(sno));
sql(2), create_clause(0), CREATE(-1)[create ], TABLE(-1)[table ], NAME(-1)[test], LBRACKET(-1)
[(], field_list(1), NAME(-1)[sno], type(0), INT(-1)[int], UNIQUE(-1)[unique],
field_list_tail(0), COMMA(-1)[,], NAME(-1)[sname], type(2), CHAR(-1)[char], LBRACKET(-1)[(],
NUMBER(-1)[9], RBRACKET(-1)[)], field_list_tail(2), COMMA(-1)[,], PRIMARY(-1)[primary ], KEY(-1)
[key], LBRACKET(-1)[(], column_list(1), column(2), NAME(-1)[sno], column_list_tail(0), NONE(-1)
[)], RBRACKET(-1)[)], field_list_tail(3), NONE(-1)[)], RBRACKET(-1)[)], NONE(-1)[;],
SEMICOLON(-1)[;] $$

>> insert into test values(1,'6');
sql(1), insert_clause(0), INSERT(-1)[insert ], INTO(-1)[into ], table_list(0), NAME(-1)[test],
table_list_tail(0), NONE(-1)[values], package_list(0), package(0), VALUES(-1)[values],
LBRACKET(-1)[(], value_list(0), value_expr(0), term(0), factor(3), NUMBER(-1)[1],
factor_tail(0), NONE(-1)[,], term_tail(1), NONE(-1)[,], value_list_tail(0), COMMA(-1)[,],
value_expr(0), term(0), factor(4), STRING(-1)['6'], factor_tail(0), NONE(-1)[)], term_tail(1),
NONE(-1)[)], value_list_tail(1), NONE(-1)[)], RBRACKET(-1)[)], package_list_tail(1), NONE(-1)
[;], SEMICOLON(-1)[;] $$
```

在输出结果中，符号名表示了语法树中的一个节点名称，圆括号内包含了由此节点向下使用的推理规则，方括号内包含了匹配字符串信息。

Compiler模块

Compiler模块将语法结构编译为字节码，并逐条运行，这一过程与数据库后端联系紧密，不易独立测试，而且模块的复杂度较低，因此没有进行专门测试样例的设计。

Engine 模块

Engine模块抽象化整个数据库后端，可以看作一个完整的C++数据库连接器，在实际设计中，这一模块的测试结果也就标志了整个数据库引擎的正确性和性能。在此项目中，设计了七个不同的引擎调用命令环境来测试整个引擎接口的正确性和性能，详细如下：

数据表元操作测试

```
TEST(EngineTest, CreateDropDatabase){
    Engine engine;
    ASSERT_TRUE(engine.CreateDatabase("root").ok());
    std::vector<Attribute> attr{Attribute("Key", intT), Attribute("Value", fixchar32T)};
    Schema tmp("firstSchema", attr.begin(), attr.end());
    EXPECT_TRUE(engine.CreateTable(tmp).ok());
    EXPECT_TRUE(engine.CloseDatabase().ok());
    EXPECT_TRUE(engine.LoadDatabase("root").ok());
    EXPECT_TRUE(engine.DropDatabase().ok());
}
```

在此过程中，测试程序创建数据引擎，并以root为名新建数据库，在此数据库中创建了简单表firstSchema，并尝试关闭连接并重新开启连接，最后删除此临时数据库。

插入查询测试

插入和查询是数据库操作中的关键核心，在此使用两个样例，分别为单条数据插入查询测试、压力数据插入查询测试。

```
TEST(EngineTest, FirstInsertQuery){
    // 创建数据库和数据表
    Engine engine;
    ASSERT_TRUE(engine.CreateDatabase("root").ok());
    std::vector<Attribute> attr{Attribute("Key", intT), Attribute("Value", fixchar32T)};
    Schema tmp("firstSchema", attr.begin(), attr.end());
    ASSERT_TRUE(engine.CreateTable(tmp).ok());

    // 插入事务
    engine.Transaction();
    // 打开主键Cursor
    EXPECT_TRUE(engine.OpenCursor("firstSchema").ok());
    EXPECT_TRUE(engine.OpenCursor("firstSchema", "Key").ok());
    // 准备数据
    auto slice = tmp.NewObject();
    slice.SetValue(0, Value(intT, new RealValue<int32_t>(7)));
    slice.SetValue(1, Value(fixchar32T, std::string("values")));
    Value key = Value(intT, new RealValue<int32_t>(7));
    // 匹配数据页并插入
    EXPECT_TRUE(engine.PrepareMatch(&key).ok());
    EXPECT_TRUE(engine.InsertSlice(&slice).ok());
    // 匹配数据页并查询
    EXPECT_TRUE(engine.PrepareMatch(&key).ok());
    std::vector<Slice> ret;
    SharedSlicePtr tmpSlice = nullptr;
    EXPECT_TRUE(engine.NextSlice(tmpSlice).ok() );
    ASSERT_TRUE(tmpSlice != nullptr);
    // 测试数据一致性
    EXPECT_EQ( static_cast<std::string>((*tmpSlice)[1].get<FixChar32>()) , std::string("values")
);
    EXPECT_TRUE(engine.DropDatabase().ok());
}
```

```
TEST(EngineTest, InsertQueryPressureTest){
    // 创建数据库和数据表
    Engine engine;
    ASSERT_TRUE(engine.CreateDatabase("root").ok());
    std::vector<Attribute> attr{Attribute("Key", intT), Attribute("Value", fixchar32T)};
    Schema tmp("firstSchema", attr.begin(), attr.end());
    ASSERT_TRUE(engine.CreateTable(tmp).ok());
    // 插入事务
    engine.Transaction();
    EXPECT_TRUE(engine.OpenCursor("firstSchema").ok());
    EXPECT_TRUE(engine.OpenCursor("firstSchema", "Key").ok());
    // 准备数据
    auto slice = tmp.NewObject();
    slice.SetValue(0, Value(intT, new RealValue<int32_t>(7)));
    slice.SetValue(1, Value(fixchar32T, std::string("values")));
    int size = 35000;

    for(int i = 0; i < size; i++){
        // 准备数据
        Value key(intT, new RealValue<int32_t>(i));
        slice.SetValue(0, key );
        slice.SetValue(1, Value(fixchar32T, std::string("values")+to_string(i) ) );
        // 匹配数据页并插入
        EXPECT_TRUE(engine.PrepareMatch(&key).ok());
        // EXPECT_TRUE(engine.InsertSlice(slice).ok());
        auto status = engine.InsertSlice(&slice);
        if(!status.ok())std::cout << status.ToString() << std::endl;
        ASSERT_TRUE(status.ok());
    }


    SharedSlicePtr pSlice;
    // 匹配数据页并查询，进行一致性测试
    for(int i = size-1; i >= 0; i--){
        Value key(intT, new RealValue<int32_t>(i));
        EXPECT_TRUE(engine.PrepareMatch(&key).ok());
        EXPECT_TRUE(engine.NextSlice(pSlice).ok());
        EXPECT_EQ(std::string(pSlice->GetValue(1)) ,std::string("values")+to_string(i));
    }
    // 删除数据库
    EXPECT_TRUE(engine.DropDatabase().ok());
}
```

在测试中，使用独立的查询请求，并逐一进行一致性测试。

主键索引测试

除了简单的数据查询一致性保证，数据库的效率同样在系统设计中起到重要作用，而数据库的效率主要由索引实现，在此样例中测试了主键索引的正确性，以及顺序查询的效率。

```
TEST(EngineTest, SequenceQueryTest){
    // 建立数据库和数据表
    Engine engine;
```

```
    ASSERT_TRUE(engine.CreateDatabase("root").ok());
    std::vector<Attribute> attr{Attribute("Key", intT), Attribute("Value", fixchar32T)};
    Schema tmp("firstSchema", attr.begin(), attr.end());
    ASSERT_TRUE(engine.CreateTable(tmp).ok());
    // 插入事务
    engine.Transaction();
    EXPECT_TRUE(engine.OpenCursor("firstSchema").ok());
    EXPECT_TRUE(engine.OpenCursor("firstSchema", "Key").ok());
    // 准备数据
    auto slice = tmp.NewObject();
    slice.SetValue(0, Value(intT, new RealValue<int32_t>(7)));
    slice.SetValue(1, Value(fixchar32T, std::string("values")));
    int size = 3500;

    for(int i = 0; i < size; i++){
        // 准备数据
        Value key(intT, new RealValue<int32_t>(i));
        slice.SetValue(0, key );
        slice.SetValue(1, Value(fixchar32T, std::string("values")+to_string(i) ) );
        // 匹配数据页并插入
        EXPECT_TRUE(engine.PrepareMatch(&key).ok());
        // EXPECT_TRUE(engine.InsertSlice(slice).ok());
        auto status = engine.InsertSlice(&slice);
        if(!status.ok())std::cout << status.ToString() << std::endl;
        ASSERT_TRUE(status.ok());
    }

    // 准备顺序查询下界
    SharedSlicePtr pSlice;
    Value minKey(intT, new RealValue<int32_t>(0));
    EXPECT_TRUE(engine.PrepareSequence(&minKey, nullptr).ok());
    // 匹配数据一致性
    for(int i = 0; i < size; i++){
        EXPECT_TRUE(engine.NextSlice(pSlice).ok());
        EXPECT_EQ(std::string(pSlice->GetValue(1)) ,std::string("values")+to_string(i));
    }
    // 删除数据库
    EXPECT_TRUE(engine.DropDatabase().ok());
}
```

删除测试

在本数据库系统的设计中，由于对并发连接的兼容，选择了惰性删除和懒合并的策略，因此删除数据的测试主要在于保证删除后数据不可获取。

```
TEST(EngineTest, DeleteTest){
    Engine engine;
    // 建立数据库和数据表
    ASSERT_TRUE(engine.CreateDatabase("root").ok());
    std::vector<Attribute> attr{Attribute("Key", intT), Attribute("Value", fixchar32T)};
    Schema tmp("firstSchema", attr.begin(), attr.end());
    ASSERT_TRUE(engine.CreateTable(tmp).ok());
    engine.Transaction();
```

```
    EXPECT_TRUE(engine.OpenCursor("firstSchema").ok());
    EXPECT_TRUE(engine.OpenCursor("firstSchema", "Key").ok());
    auto slice = tmp.NewObject();
    slice.SetValue(0, Value(intT, new RealValue<int32_t>(7)));
    slice.SetValue(1, Value(fixchar32T, std::string("values")));
    int size = 35000;
    // 插入数据
    for(int i = 0; i < size; i++){
        Value key(intT, new RealValue<int32_t>(i));
        slice.SetValue(0, key );
        slice.SetValue(1, Value(fixchar32T, std::string("values")+to_string(i) ) );
        EXPECT_TRUE(engine.PrepareMatch(&key).ok());
        // EXPECT_TRUE(engine.InsertSlice(slice).ok());
        auto status = engine.InsertSlice(&slice);
        if(!status.ok())std::cout << status.ToString() << std::endl;
        ASSERT_TRUE(status.ok());
    }

    SharedSlicePtr pSlice;
    // 逐一删除数据
    for(int i = size-1; i >= 0; i--){
        Value key(intT, new RealValue<int32_t>(i));
        EXPECT_TRUE(engine.PrepareMatch(&key).ok());
        EXPECT_TRUE(engine.NextSlice(pSlice).ok());
        EXPECT_TRUE(engine.DeleteSlice(&(*pSlice)).ok());
    }
    // 逐一测试是否可获取
    for(int i = size-1; i >= 0; i--){
        Value key(intT, new RealValue<int32_t>(i));
        auto status = engine.PrepareMatch(&key); // is NOT ok
        EXPECT_TRUE(engine.NextSlice(pSlice).ok());
        EXPECT_TRUE(pSlice == nullptr);
    }
    // 删除数据库
    EXPECT_TRUE(engine.DropDatabase().ok());
}
```

非主键索引测试

在非主键索引测试中，主要完成了非主键索引的建立，索引目标正确性测试，索引的删除。通过这三个基本测试，非主键索引的功能也就得到了基本保证。

```
TEST(EngineTest, CreateNonPrimaryIndex){
    Engine engine;
    // 创建数据库和数据表
    ASSERT_TRUE(engine.CreateDatabase("root").ok());
    std::vector<Attribute> attr{Attribute("Key", intT), Attribute("Value", fixchar32T)};
    Schema tmp("firstSchema", attr.begin(), attr.end());
    tmp.SetUnique(1);
    ASSERT_TRUE(engine.CreateTable(tmp).ok());
    engine.Transaction();
    EXPECT_TRUE(engine.OpenCursor("firstSchema").ok());
    EXPECT_TRUE(engine.OpenCursor("firstSchema", "Key").ok());
```

```
    auto slice = tmp.NewObject();
    slice.SetValue(0, Value(intT, new RealValue<int32_t>(7)));
    slice.SetValue(1, Value(fixchar32T, std::string("values")));
    int size = 3500;
    // 插入数据
    for(int i = 0; i < size; i++){
        Value key(intT, new RealValue<int32_t>(i));
        slice.SetValue(0, key );
        slice.SetValue(1, Value(fixchar32T, std::string("values")+to_string(i) ) );
        EXPECT_TRUE(engine.PrepareMatch(&key).ok());
        // EXPECT_TRUE(engine.InsertSlice(slice).ok());
        auto status = engine.InsertSlice(&slice);
        if(!status.ok())std::cout << status.ToString() << std::endl;
        ASSERT_TRUE(status.ok());
    }
    // 建立非主键索引
    auto status = engine.MakeIndex("firstSchema", "Value", "secondIndex");
    if(!status.ok())std::cout << status.ToString() << std::endl;
    EXPECT_TRUE(status.ok());
    // 打开非主键查询句柄
    EXPECT_TRUE(engine.OpenCursor("firstSchema", "Value").ok());
    SharedSlicePtr pSlice;
    int test = 374;
    Value key(fixchar32T, std::string("values")+to_string(test));
    status = engine.PrepareMatch(&key);
    if(!status.ok())std::cout << status.ToString() << std::endl;
    EXPECT_TRUE(status.ok());
    bool found = false;
    // 测试查询回调正确性
    while(true){
        status = engine.NextSlice(pSlice);
        if(!status.ok())cout << status.ToString() << endl;
        EXPECT_TRUE(status.ok());
        if(!pSlice)break;
        if(std::string(pSlice->GetValue(0)) == to_string(test))found = true;
    }
    EXPECT_TRUE(found);
    // 删除非主键索引
    status = engine.DropIndex("firstSchema", "secondIndex");
    if(!status.ok())std::cout << status.ToString() << std::endl;
    EXPECT_TRUE(status.ok());
    // 删除数据库
    EXPECT_TRUE(engine.DropDatabase().ok());
}
```

测试结果

全部通过：

```
Running main() from gtest_main.cc
[==========] Running 7 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 7 tests from EngineTest
```

```
[ RUN      ] EngineTest.CreateDropDatabase
[       OK ] EngineTest.CreateDropDatabase (32 ms)
[ RUN      ] EngineTest.FirstInsertQuery
[       OK ] EngineTest.FirstInsertQuery (17 ms)
[ RUN      ] EngineTest.InsertQueryPressureTest
[       OK ] EngineTest.InsertQueryPressureTest (2708 ms)
[ RUN      ] EngineTest.SequenceQueryTest
[       OK ] EngineTest.SequenceQueryTest (197 ms)
[ RUN      ] EngineTest.CreateNonPrimaryIndex
[       OK ] EngineTest.CreateNonPrimaryIndex (238 ms)
[ RUN      ] EngineTest.DeleteTest
[       OK ] EngineTest.DeleteTest (3863 ms)
[ RUN      ] EngineTest.DropNonPrimaryIndex
[       OK ] EngineTest.DropNonPrimaryIndex (308 ms)
[----------] 7 tests from EngineTest (7462 ms total)

[----------] Global test environment tear-down
[==========] 7 tests from 1 test case ran. (7487 ms total)
[  PASSED  ] 7 tests.
```

Page Manager 模块

Page Manager为全局操作的最终承担者，它的安全性决定了整个系统的运行时状态，但由于它提供的接口有限，而更复杂的同步接口难以独立测试，在本设计中只对基本的操作系统相关接口进行独立测试：

三个测试，分别测试了文件创建与删除接口、分页创建与读写接口和数据一致性、分页创建删除的压力测试。

```
TEST(PageManagerTest, FileTest){
    PageManager pager;
    // 创建文件测试
    for(int i = 0; i<10; i++){
        string name = string("pager_test")+to_string(i);
        FileHandle ret;
        EXPECT_TRUE(pager.NewFile(name, 4, ret).ok());
        EXPECT_TRUE(pager.DeleteFile(ret).ok());
    }
}

TEST(PageManagerTest, PageTest){
    size_t len = 4096;
    PageManager pager;
    FileHandle file;
    // 创建文件测试
    ASSERT_TRUE(pager.NewFile("test_io", 4, file).ok());

    char* data = new char[len];
    memset(data, 1, len);
    PageHandle page;
    Status status;
    // 创建分页
    status = pager.NewPage(file,kBFlowPage, page);
    if(!status.ok()){
        cout << status.ToString() << endl << flush;
```

```
        exit(0);
    }
    // 写入分页
    status = pager.DirectWrite(page, data);
    if(!status.ok()){
        cout << status.ToString() << endl << flush;
        exit(0);
    }
    // 关闭文件
    ASSERT_TRUE(pager.CloseFile(file).ok());
    // 打开文件
    ASSERT_TRUE(pager.OpenFile("test_io", file).ok());
    page = GetPageHandle(file, 1);
    char* ret;
    // 读取文件并测试一致性
    PageRef* ref = new PageRef(&pager, page, kReadOnly);
    ret = ref->ptr;
    for(int i = 0; i<len; i++)EXPECT_EQ(data[i], ret[i]);
    delete ref;
    delete [] data;
    ASSERT_TRUE(pager.DeleteFile(file).ok());
}
TEST(PageManagerTest, PressureTest){
    size_t len = 4096;
    PageManager pager;
    FileHandle file;
    // 创建文件
    ASSERT_TRUE(pager.NewFile(string("pressure"), 4, file).ok());
    // 创建分页
    for(int i = 0; i < 998; i++){
        PageHandle page;
        ASSERT_TRUE(pager.NewPage(file, kBFlowPage, page).ok());
        // 同步至内存
        auto ret = pager.Pool(page);
        ASSERT_TRUE(ret.ok());
    }
    // 删除分页
    for(int i = 0; i < 100; i++){
        EXPECT_TRUE(pager.DeletePage(i+1).ok());
    }
    // 删除文件
    auto ret = pager.DeleteFile(file);
    EXPECT_TRUE(ret.ok());
}
```

3.2 综合测试

在基础的独立模块测试之上，系统整体的测试以更复杂的运行时逻辑对它们之间的交互和运作进行测试，下面以不同类型的SQL语句来测试MiniSQL引擎的功能。

3.2.1 功能测试

1. **测试create table功能**，测试用例如下：

```
// 正确建表
create table student (
id int,
name char(10),
score float,
primary key (id)
);

// 正确建表，并进行unique约束
create table book (
id int,
name char(20) unique,
price float,
primary key (id)
);

// 测试对于重复建表是否能够返回错误
create table student (
id int,
name char(10)
);

// 测试对于表中属性重复是否能够返回错误
create table teacher(
name char(10),
name char(20),
salary int
);
```

3. **测试execfile的功能**，执行：

```
execfile student.sql;
// student.sql
create table student(
sno int,
primary key(sno)
);
```

4. **测试insert功能**，对于insert功能的测试可以结合**对select部分功能的测试**，执行如下语句，检查输出的记录是否和插入的记录相同：

```
>> insert into student
values(1,'a'),
values(2,'b'),
values(3,'c');

>> select * from student;
```

5. **测试select的功能**，测试用例如下：

```
//测试对于int类型的等值查询
select * from student where id = 125346;

//测试对于char(n)类型的等值查询
select * from student where name = "nick";

//测试对于float类型的等值查询
select * from student where score = 95.5;

// 测试char(n)类型的模糊查询
select * from student where name like '%a';

//测试对于int类型的区间查询
select * from student where id <> 111111;

//测试对于char(n)类型的区间查询
select * from student where name <= "Lily";

//测试对于float类型的区间查询
select * from student where score > 92;

//测试同一属性上的多条件查询
select * from student where score >= 90 and score < 100;

//测试不同属性上的多条件查询
select * from student where id > 222222 and score <= 89;

//测试在不存在的表中进行查询是否会报错
select * from teacher;

//测试where条件涉及不存在的属性是否会报错
select * from student where age = 20;
```

5. **测试delete功能**，测试用例如下：

```
//测试能否删除
delete from student where id = 222222;
select * from student where id = 222222;

//测试删除不存在的记录
delete from student where id = 222222;

//测试插入已经删除的记录
insert into student (222222, Bryant, 81);

//测试删除整个表中的记录
delete from student;
select * from student;
```

6. **测试create index功能**，测试用例如下：

```
//测试能否正确建立索引
create index nameidx on book(name);

//测试能否检查重复建立索引
create index nameidxidx on book(name);

//测试对非unique属性建立索引能否返回错误
create index priceidx on book(price);

//测试在不存在的表或属性上建立索引能否返回错误
create index nameidxidxidx on teacher(idx);
```

9. **测试drop index功能**，因为之前已经测试了create index检查重复建立索引的功能，所以这里可以利用这一点来测试，测试用例如下：

```
//测试能否正确删除索引
drop index nameidx;
create index nameidx on book(name);

//测试删除不存在的索引是否会返回错误
drop index nameidxidxidx;
```

10. **测试drop table功能**，测试用例如下：

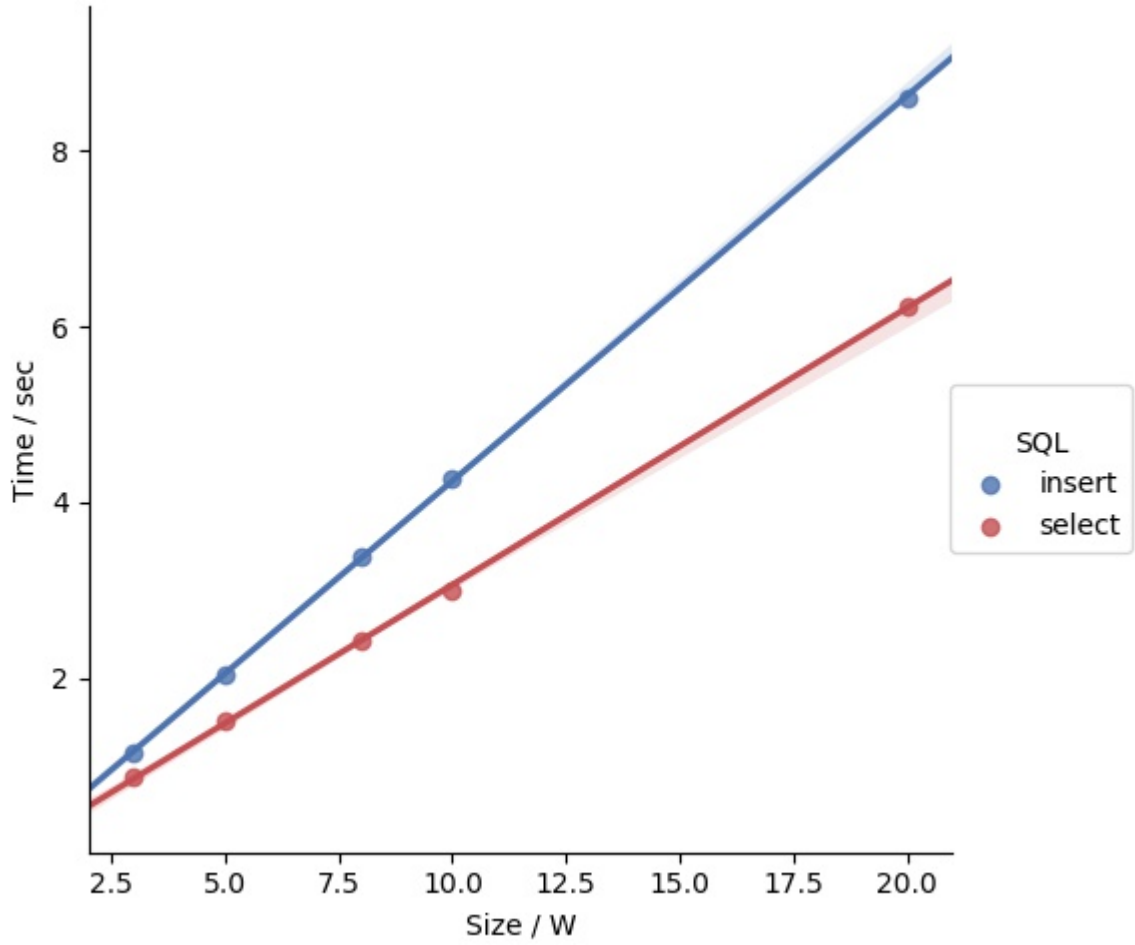
```
//测试能否正确删除表
drop table student;
select * from student;

//测试删除不存在的表是否会返回错误
drop table student;
```

至此，整个数据库系统的设计要求已经全部经过测试，系统的功能得到了基本保证。

性能测试

在性能测试中，结合Engine类裸接口和Compiler前端接口进行多层次测试对比。

首先进行 **Engine裸接口测试**，测试压力插入数据和压力查询数据时间，这里使用的测试语境为独立语句，独立事务，无额外索引，测试结果如下：



耗时曲线呈现完美的线性关系，没有出现大规模数据下的性能下降情况。同时整体性能达到了比较好的水平。

接下来进行整合前端的 **Database外部接口测试**，在此测试中运行完整的语法分析过程，以文件读入为指令输入模式。

对十万数据进行测试，结果如下：

```
>> execfile insert-pressure.sql;
Front-end: 174.635
Back-end: 6.42862
```

可以看到，后端时间与裸测试相符，而近乎全部时间用于语法分析和编译。

下面简单对这一现象进行分析：

前端运行过程分为四个部分，通过分段计时，将它们的分别耗时情况记录如下：

- 读取语句: -
- 正则匹配：112.048 s
- 语法结构分析：52.599 s
- 编译和包装：8.46 s

可以明显地看到，C++11糟糕的原生正则引擎库使整个前端过程性能下降了两倍。同时，为了实现对复杂SQL语句和条件从句的兼容，使用了较为复杂的上下文无关文法，这使得语法分析阶段耗时较多。

基于以上的分析，这一MiniSQL系统的未来优化将专注于前端正则匹配的重新编写，以及对语法规则的精简，和推理逻辑的优化。