**实验2**

**date**

* 1. 实验要求

在现有功能上实现日期类型字段。

当前已经支持了int、char、float类型，在此基础上实现date类型的字段。date测试可能超过2038年2月，也可能小于1970年1月1号。注意处理非法的date输入，需要返回FAILURE。

这道题目需要从词法解析开始，一直调整代码到执行阶段，还需要考虑DATE类型数据的存储。

注意：

- 需要考虑date字段如何比较大小;

- 这里没有限制日期的范围，需要处理溢出的情况。

- 需要考虑闰年。

* 1. CREATE TABLE语句的执行过程（1.5h）

本节内容不涉及代码更改。

本节内容分析miniob中建表语句的执行过程，探究建表语句会涉及到的模块，以加深同学们对miniob整体的理解。

date字段主要涉及CREATE TABLE建表语句以及SELECT查询语句，初次接触时，我们可以观察这些语句的执行逻辑，进而理解要增加新的数据类型date时有哪些位置的代码需要修改。

本节内容对下述简单建表语句进行分析。

CREATE TABLE test(id int, age int)

正如【实验1：basic】中所述，miniob对SQL语句的执行分为5个阶段，在session\_stage.cpp中提供了五个阶段的接口，我们以此为基础展开分析。

* + 1. query cache阶段与parse 阶段

首先在session\_stage.cpp中打上断点，开始调试：

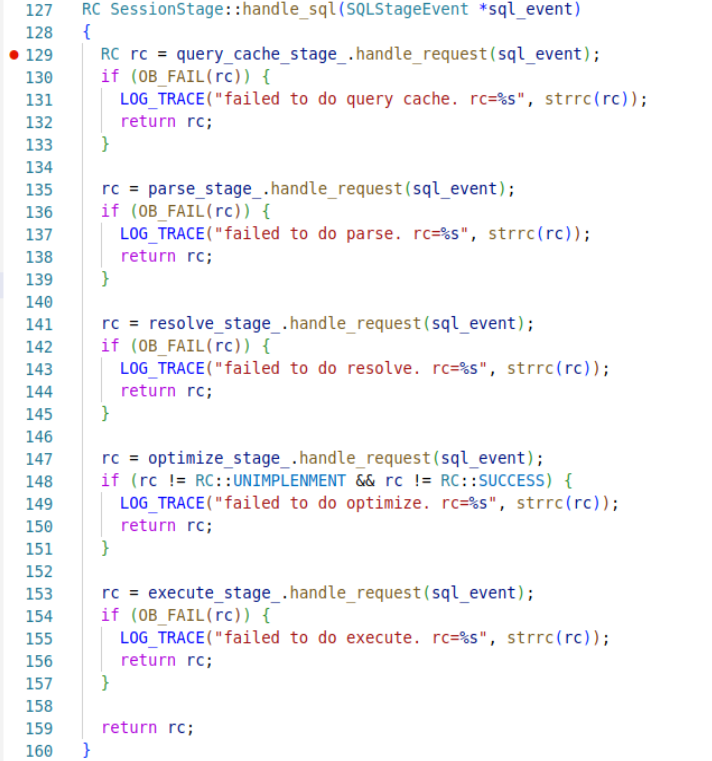


图1 session\_stage.cpp 断点调试

几乎所有的SQL语句都可以在这里打断点进行调试，在进入该函数之前大多是环境初始化、建立会话、服务器与客户端通信初始化的相关代码。感兴趣的同学可以尝试研究之前的代码执行流程。

SQL语句的执行在miniob中被划分为5个阶段，**首先是query cache阶段**。但是miniob的query cache阶段仅仅留作了接口，并未作实现：

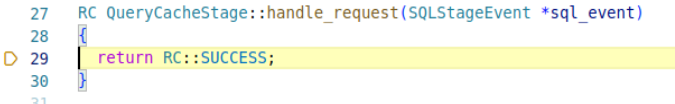


图2 query\_cache.cpp

RC::SUCCESS是全局定义的返回状态码（Retrun Code），方便对不同种类的错误分类，在rc.h中定义。

随后进入parse\_stage阶段。在该阶段我们拿到std::string类型的输入SQL语句，调用词法分析、语法分析的接口函数parse()，解析的结果保存在ParsedSqlResult类的实例中。具体流程如下图所示：

文本

描述已自动生成

图3 parse\_stage.cpp

可以看出，最核心的部分为第39行的parse(…)函数以及第37行用于保存解析结果的ParsedSqlResult类。首先查看ParsedSqlResult类的源码：

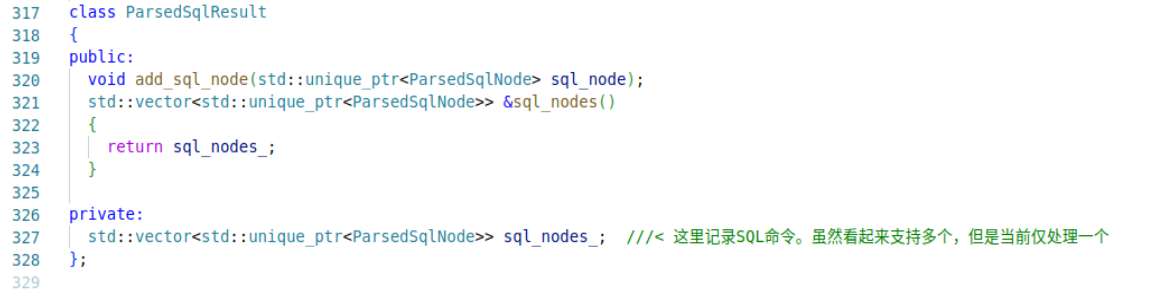


图4 parse\_defs.h

在ParsedSqlResult类中，327行的sql\_nodes数组中存储了多个ParsedSqlNode类型的指针，ParsedSqlNode类型用于存储每一个解析后的SQL命令。尽管这里采用了vector数组来存储，看起来是支持多个SQL命令的，但在目前的miniob中，仅仅支持处理一个SQL命令。接着，我们查看ParsedSqlNode类的源码：

图片包含 表格

描述已自动生成

图5 parse\_defs.h

ParsedSqlNode类的成员变量由各种类型的“SqlNode”组成（293~306行）。通过parse(…)函数后，ParsedSqlNode类中对应语句的成员变量中会记录相关的信息，而语句类型则是通过292行枚举类型SqlCommandFlag定义。

若想增加新的SQL语句类型，应当增加对应语句的SqlNode类。

在本例中，经过parse(…)函数后，CreateTableSqlNode类的实例create\_table中将会保存表名、属性名以及属性类型。我们查看CreateTableSqlNode类的定义：

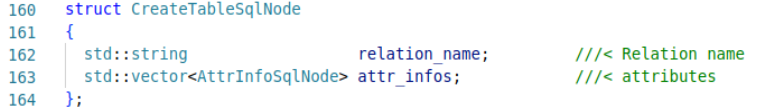


图6 parse\_defs.h

可以看出，表名采用std::string类型存储，而属性名及对应的数据类型则在AttrInfoSqlNode类中定义。我们查看AttrInfoSqlNode类的定义：

文本

描述已自动生成

图7 parse\_defs.h

同样，属性名也是采用std::string类型存储，属性的类型则属于AttrType，这是一个枚举变量。属性长度可以暂时不关心，这涉及到了底层的存储逻辑。我们查看AttrType的定义：

文本

描述已自动生成

图8 value.h

可以看到，miniob支持的全部数据类型都在这里定义了。

如果要增加一个新的数据类型的话，首先应该在这里添加新类型的枚举定义。当然，为了增加DATE字段，也应当在这里增加新的类型DATES。

接下来，我们分析parse(…)函数是如何解析输入的SQL语句的。

* + 1. 词法分析与语法分析

SQL解析的第一步是词法分析与语法分析，原理与编译技术中介绍的类似。客户端的输入是一个字符串SQL指令，词法分析与语法分析解决了如何对字符串分词、分词后的每个子串的作用以及字串中的信息如何处理的问题。本实验课的重点是如何拓展MiniOB的SQL解析部分，使之可以支持我们想要增加的新语法与新功能。

* + - * 词法分析

词法分析的基础功能是以字符为单位，读取输入的字符串（输入的SQL命令），然后按照特定的规则模式去匹配相应的字符串。

表格

描述已自动生成

图9 lex\_sql.l

上图是词法分析文件的部分代码，每一行左侧表示匹配模式，右侧表示返回的token类型，token类型在语法分析的.y文件中定义，它标明了当前匹配到的子串所述的类型。比如，88行左侧定义的“CREATE”表示对“CREATE”字符串的完全匹配， 80行左侧的模式表示一个数字的正则表达式。当输入的SQL语句中出现了CREATE字符串时，词法分析器便会返回CREATE token；当输入的SQL语句中出现了一个数字时，词法分析器便会返回NUMBER token。这些特定的token会在语法分析阶段发挥作用。

拓展miniob的SQL解析部分时，我们只需要在lex\_sql.l文件中增加新的词法匹配模式。即如果要增加date类型的话，也需要在该文件中增加新的词法模式。

* + - * 语法分析

语法分析的基础功能是根据词法分析的结果，按照特定的语法规则，生成抽象语法树。我们可以在yacc\_sql.y文件中看到下面的代码：

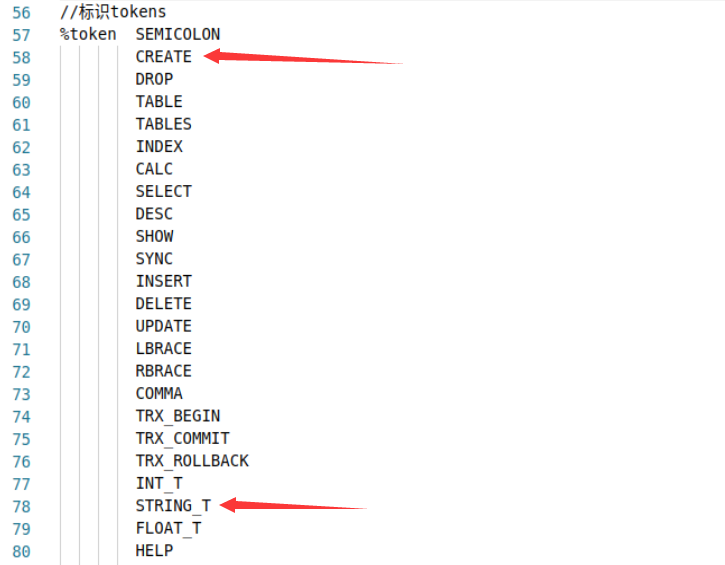


图10 yacc\_sql.y

这里定义了解析过程中可以识别的全部token类型，也对应了图9所示的lex\_sql.l文件中每一个模式返回的token类型。

这一部分可以结合lex\_sql.l文件阅读。增加date类型，需要在这里增加新的token类型后，再在lex\_sql.l中增加匹配模式

在yacc\_sql.y中可以看到下面的代码：

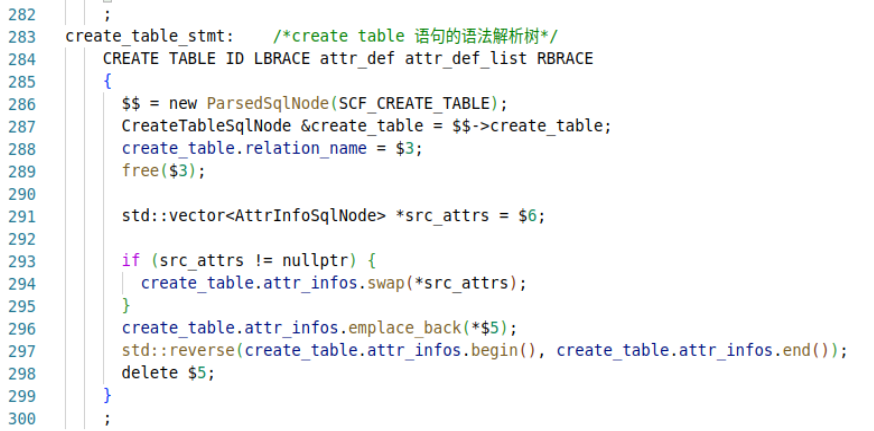


图11 yacc\_sql.y

这是对语法规则的自定义描述，具体来说，是对CREATE TABLE建表语句的语法规则描述。283行与284行定义了哪些类型的token的组合（CREATE TABLE ID LBRACE attr\_def attr\_def\_list RBRACE）构成了建表语句（create\_table\_stmt）。具体来说，当词法分析器依次解析出了token CREATE、token Table、token ID、token LBRACE、token attr\_def、token attr\_def\_list、token RBRACE时，便会认为当前的SQL命令是建表语句create\_table\_stmt。token CREATE在上文中已经出现过了，我们来观察token ID是在哪里定义的：

图片包含 文本

描述已自动生成

表格

中度可信度描述已自动生成

图12 lex\_sql.l

结合65行与117行可以看出，token ID表示由大小写字母、数字和下划线组成的且开头不能是数字的字符串。同理可以在lex\_sql.l中查找其他token的含义。

接着分析create\_table\_stmt的语法规则。当匹配到这些token组合时，会自动的执行285行~299行的C/C++代码段。$$表示语法规则左侧即create\_table\_stmt变量，$1表示CREATE对应的变量，$2表示TABLE对应的变量，$3、$4、$5同理。这些$符号所指代的变量的类型在之前也给出了定义：

文本

描述已自动生成

图13 yacc\_sql.y

图形用户界面, 文本

描述已自动生成

图14 yacc\_sql.y

在图13的149行，指出了create\_table\_stmt是sql\_node类型，而在图14的103行指出，sql\_node即ParsedSqlNode\*类型，该类型我们在上一节内容中已经提到过了。$3、$5、$6所指代的变量类型同理，分别是std::string、AttrInfoSqlNode\*、std::vector<AttrInfoSqlNode>\*类型（$1、$2等没有提到的则意味着无需定义类型，只是作为语法规则的一部分即可）。

当我们添加新的date类型时，除了增加了词法分析中的模式、语法分析中声明返回的token类型外，可能还需要指定该token类型对应的C/C++变量类型。

再举一个例子。下面的代码段是token type的语法规则定义。图15的339行与340行表示token INT\_T在语法分析阶段会被视作token type，同时，图16的128行与117行共同定义了token type的C/C++变量类型是int类型，因此，在匹配到token INT\_T的同时，token type会被赋值为INTS，INTS即前文中提到的、在value.h中定义的枚举类型。

文本

低可信度描述已自动生成

图15 yacc\_sql.y

图形用户界面, 文本

描述已自动生成

图16 yacc\_sql.y

图15的341行与342行表示token type还可以标识的其他类型。

与词法分析类似，我们在拓展语法分析时，最关心的是规则的编写部分，同样，增加date类型也需要增加新的语法规则。

接下来我们来介绍CREATE TABLE语句对应的parse(…)函数具体如何执行。

* + 1. CREATE TABLE的语法解析

我们再来回到本节的例子：

CREATE TABLE test(id int, age int)

根据lex\_sql.l中定义的词法规则，上述SQL字符串会被解析为下图所示的抽象语法树：

图示

描述已自动生成

图17 抽象语法树

对照抽象语法树，我们可以分析parse(…)函数的执行过程：

将输入的SQL语句视为字符串，依次扫描每个字符；当扫描了前6个字符时，词法分析器匹配到了模式CREATE，因此返回给语法分析器一个token CREATE；同理，继续顺序扫描，依次归约出token TABLE、token ID、token LBRACE、token ID、token INT\_T，此时语法分析部分的缓冲区如下：

CREATE TABLE ID LBRACE ID INT\_T

最右端的token INT\_T，可以根据下图语法规则，归约出token type。同时，int类型的token type被赋值为枚举值INTS。

文本

低可信度描述已自动生成

图18 yacc\_sql.y

此时语法分析缓冲区如下：

CREATE TABLE ID LBRACE ID type

最右端的token ID、token type，可以根据下图的语法规则，归约出token attr\_def。同时，AttrInfoSqlNode\*（该类型在parse\_defs.h中定义，是一个结构体）类型的token attr\_def的相关成员变量被赋值——AttrInfoSqlNode.type被赋值为$2即token type的值，AttrInfoSqlNode.name被赋值为$1即ID的值（字符串test），AttrInfoSqlNode.length被赋值为4。

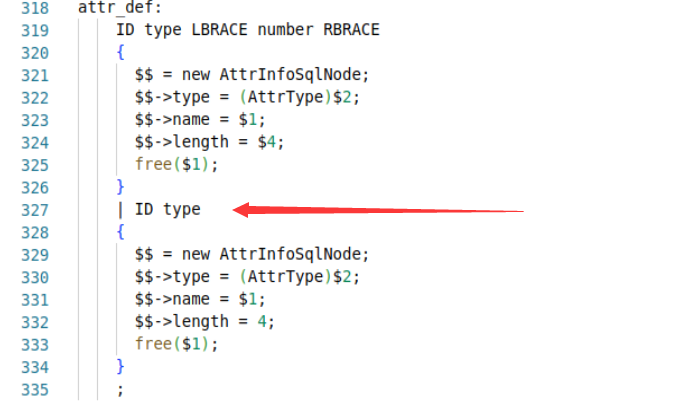


图19 yacc\_sql.y

此时语法分析缓冲区如下：

CREATE TABLE ID LBRACE attr\_def

词法分析器继续顺序扫描输入字符串，语法分析缓冲区如下：

CREATE TABLE ID LBRACE attr\_def COMMA ID INT\_T

同理，最右端的token INT\_T，可以归约出token type。

CREATE TABLE ID LBRACE attr\_def COMMA ID type

同理，最右端的token ID、token type，可以归约出token attr\_def。

CREATE TABLE ID LBRACE attr\_def COMMA attr\_def。

此时，最右端出现了token COMMA、token attr\_def，考虑如下规则：

图形用户界面, 文本, 应用程序

描述已自动生成

图20 yacc\_sql.y

如果语法分析缓冲区中再出现一个token attr\_def\_list，即可与前面的token COMMA、token attr\_def归约；又考虑到attr\_def\_list可以推导出**EMPTY**，因此，语法分析器会自动添加一个token EMPTY到缓冲区：

CREATE TABLE ID LBRACE attr\_def COMMA attr\_def EMPTY

并将右侧的token COMMA、token attr\_def、token EMPTY按照上述规则归约：

CREATE TABLE ID LBRACE attr\_def attr\_def\_list

随后，词法分析器继续顺序扫描输入的SQL字符串，得到token RBRACE：

CREATE TABLE ID LBRACE attr\_def attr\_def\_list RBRACE

此时，缓冲区中的token符合如下规则：

图形用户界面, 文本, 应用程序

描述已自动生成

图21 yacc\_sql.y

对缓冲区中的token采用该规则归约，得到token create\_table\_stmt，并执行大括号内的动作（C/C++代码，自行对照lex\_sql.l、yacc\_sql.y、parse\_defs.h查看理解）此时缓冲区如下：

create\_table\_stmt

随后，会根据如下规则，归约出token command\_wrapper；再根据其他已有规则，归约出开始符号commands。

文本

低可信度描述已自动生成

图22 yacc\_sql.y

这里提到的其他已有规则可以不必深究，大多是一些控制SQL输入串格式的规则，包括多行SQL语句等。至此，parse阶段结束，建表语句涉及到的表明、属性列表等已经正确存储在了ParsedSqlNode类中，以供后续阶段使用。

* + 1. resolve阶段

图形用户界面, 文本, 应用程序

描述已自动生成

图23 session\_stage.cpp

parse阶段结束后，进入resolve阶段。在resolve阶段，前一阶段输出的ParsedSqlNode会被转换为标准的Stmt结构，同时，进行错误检查。

继续分析建表语句的例子：

CREATE TABLE test(id int, age int)

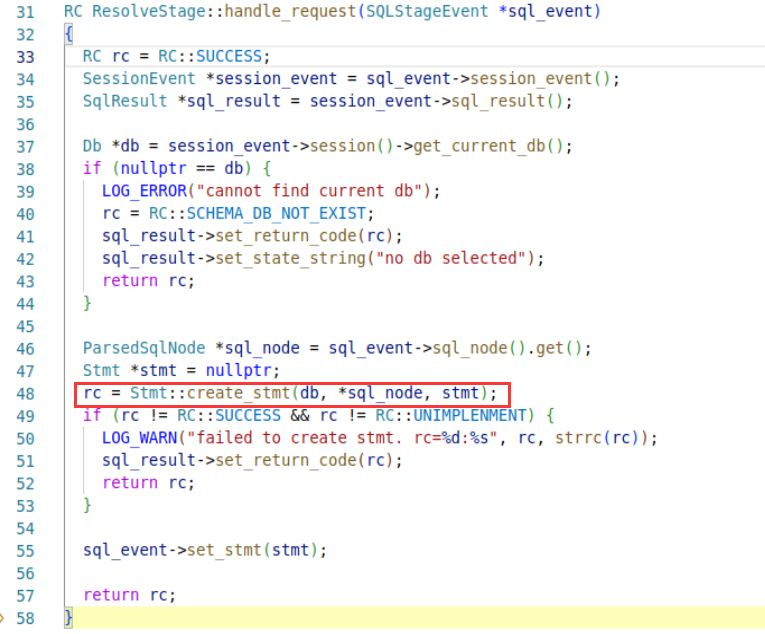


图24 resolve\_stage.cpp

与parse阶段的接口类似，resolve接口函数中33行~44行不需要太过关心，进行了一些必要的准备工作。可以发现，最关键的是48行，将数据库db、parse阶段的输出sql\_node、以及resolve阶段的结果指针stmt作为参数，创建标准的Stmt。我们跳转到该函数：

图片包含 日程表

描述已自动生成

图25 stmt.cpp

由于我们输入的是CREATE TABLE语句，此时sql\_node.flag为SCF\_CREATE\_TABLE，因此程序会跳转到57行：

文本

低可信度描述已自动生成

图26 create\_table\_stmt.cpp

CreateTableStmt类的定义如下：

图形用户界面, 文本, 应用程序

描述已自动生成

图27 create\_table\_stmt.h

可以看出，CREATE TABLE语句的resolve阶段是将sql\_node中保存的关系名、字段名与字段类型转换成了CreateTableStmt类型并返回，作为下一阶段的输入。

如果要增加一个新的SQL语句类型，我们需要创建对应类型的Stmt类。

不过，CREATE TABLE语句不涉及到错误检查，我们以INSERT语句为例，下面是INSERT语句在resolve阶段的接口函数：

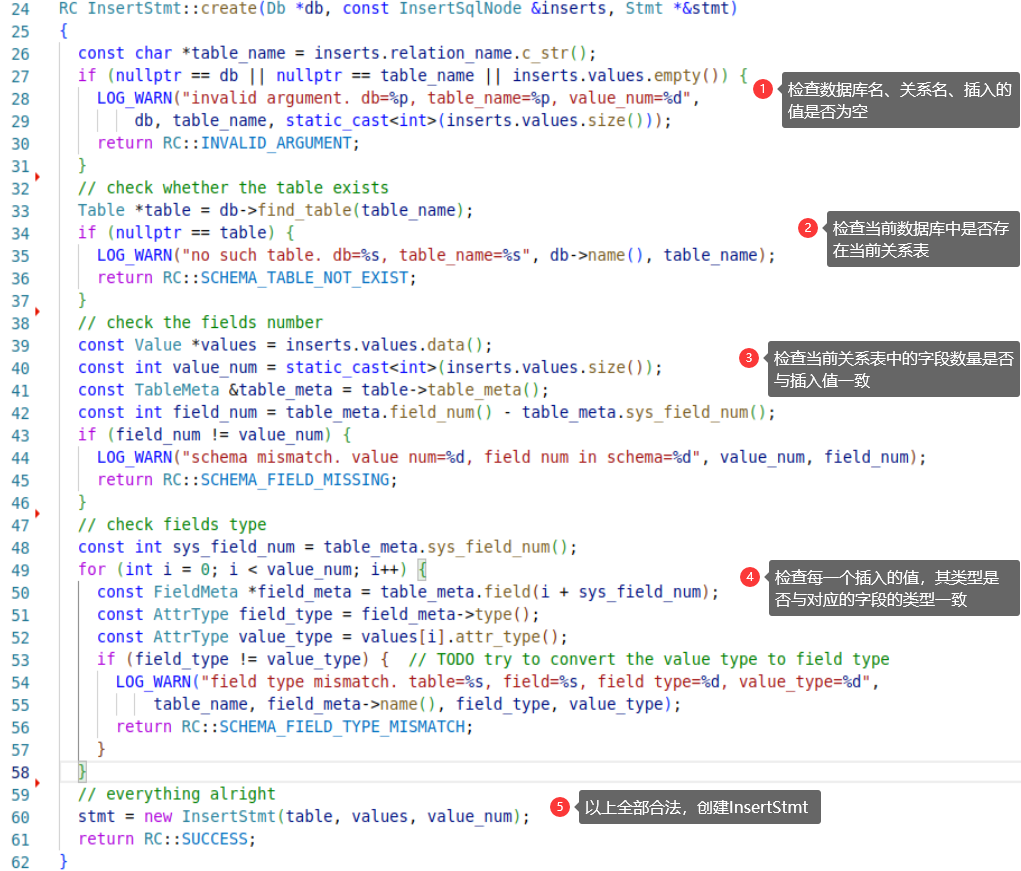


图28 insert\_stmt.cpp

可以看出，INSERT语句的resolve阶段进行了错误检查，分别检查了数据库名、关系表名、插入的值是否为空，当前数据库中是否存在当前的关系表，当前关系表中的字段数量是否与插入的数量一致，每一个插入的值的类型是否与对应字段的类型一致。

* + 1. optimize阶段

optimize阶段会为Stmt创建逻辑计划与物理计划，对于CREATE TABLE语句而言，无需optimize阶段，关于optimize阶段会在【实验3 update】中介绍。

图形用户界面, 文本, 应用程序

描述已自动生成

图29 logical\_plan\_generator.cpp

* + 1. execute阶段

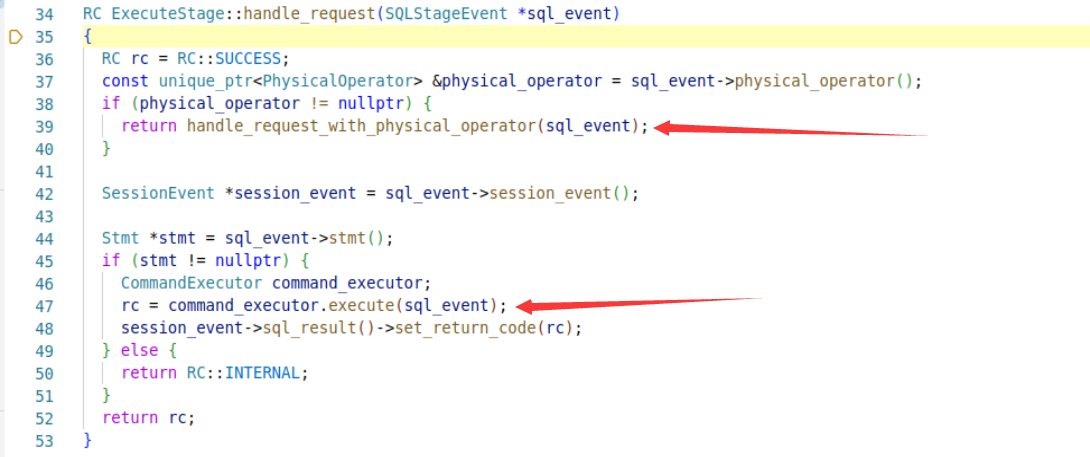


图30 execute\_stage.cpp

execute阶段的接口如图30。可以看出，如果在optimize阶段创建了物理计划，则按照物理计划执行（39行）；如果没有物理计划的话，则跳入47行执行。我们查看47行函数的源码：

图形用户界面, 文本, 应用程序

描述已自动生成

图31 command\_executor.cpp

跳转41行：

文本

描述已自动生成

图32 create\_table\_executor.cpp

可以看出，在execute阶段，调用了数据库的create\_table(…)接口，完成了新关系表的创建。

如果要增加一个新的SQL语句类型，我们需要创建对应类型的Executor类。

* 1. 实现思路（0.5h）

date字段除了与建表语句相关外，还与insert语句相关。参考CREATE TABLE语句的流程，同学们请自行观察INSERT语句的执行流程，加深对miniob的理解。

INSERT语句的语法分析中，如图33所示，出现了value与value\_list两种token。

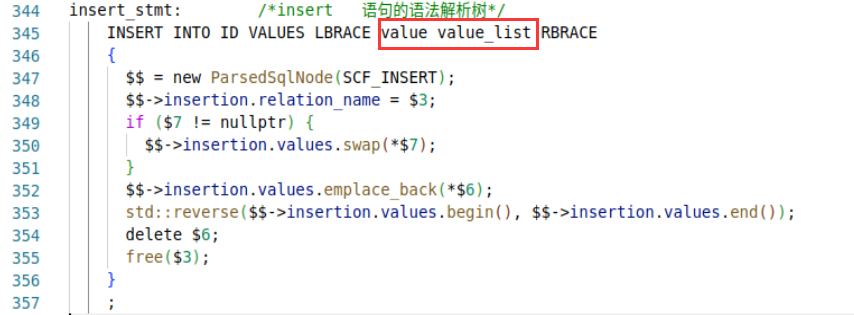


图33 yacc\_sql.y

我们查看value token的语法规则：

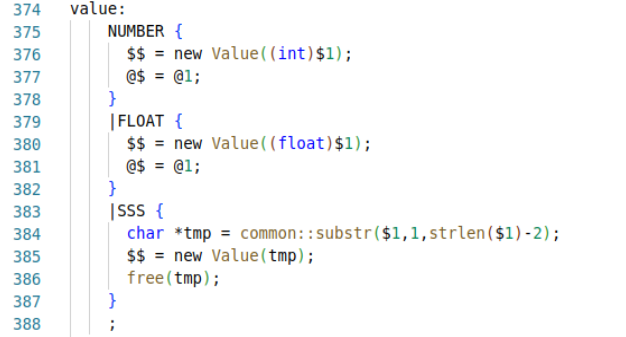


图34 yacc\_sql.y

容易看出，目前miniob支持的三种数据类型均被解析成value token。在解析的过程中，创建Value类的变量用于保存相关值。因此，同学们可以从Value类入手，逐步实现date字段。

我们给出date字段的实现步骤：

1. 学习flex工具支持的正则表达式语法，为date字符串编写词法匹配模式。
2. 修改lex\_sql.l、yacc\_sql.y、parse\_defs.h，使miniob支持date字段相关语法。
3. 考虑date类型的数据的存储结构，在实现上即可以采用字符串存储，也可以采用int类型存储。
4. 索引可以被建立在date类型的字段上，因此还需实现date类型间的比较。
5. 考虑非法输入情况。

具体的实现方式可以参考第4节内容。

* 1. date类型的实现（1.5h）

我们采用int类型存储日期字符串，在查询其值时，再将int类型转换为YYYY-MM-DD的日期字符串类型即可。

* + 1. 词法、语法修改

仿照int类型，我们在lex\_sql.l中增加新的模式，以匹配DATE字段与日期字符串。

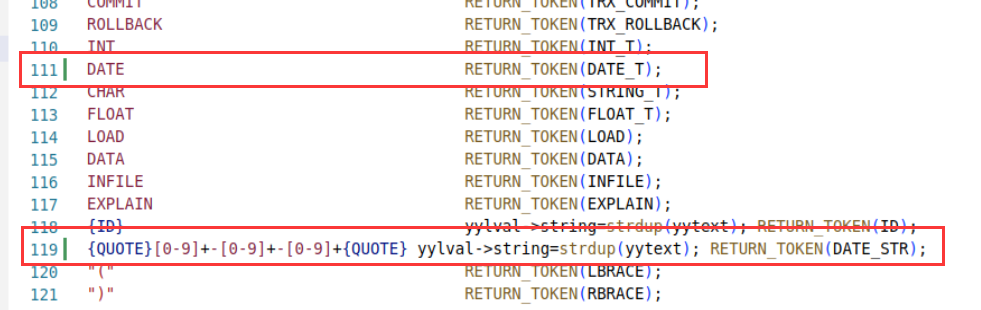


图35 lex\_sql.l

此时，我们引入了两个新的token：DATE\_T与DATE\_STR，需要在yacc\_sql.y中声明：

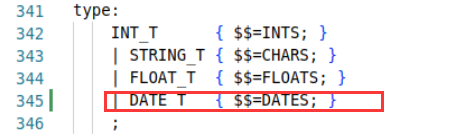


文本

描述已自动生成

图36 yacc\_sql.y

同时，在yacc\_sql.y中，仿照int与string类型的实现方法，我们还需要增加新的语法规则：

图形用户界面, 文本, 应用程序

描述已自动生成

图37 yacc\_sql.y

可以看出，我们需要找到INTS、CHARS、FLOATS这些枚举类型的定义，并在其中加入DATES类型的定义；还需要找到Value类的定义，为其添加新的构造函数(393行)。

* + 1. Value类的修改

在value.h中，找到枚举类型AttrType，增加DATES类型：

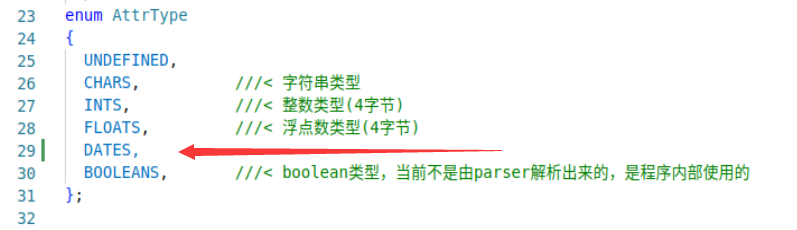


图38 value.h

观察Value类的成员函数，不难发现，miniob中对数据类型的大多数操作都在Value类中实现，对照其他数据类型的实现方式，我们定义以下函数，并在value.cpp中实现。

explicit Value(const char \*date, int len, int flag);

void set\_date(int val);

int get\_date() const;

union {

int int\_value\_;

float float\_value\_;

bool bool\_value\_;

int date\_value\_; //采用int存储date类型

} num\_value\_;

特别的，构造函数中的int flag参数是为了区别于普通的字符串类型。在value.cpp中实现上述函数时，浏览value.cpp中其他函数的实现，可以发现现有的如get\_int(…)等函数中包含了类型强制转换，因此value.cpp中需要修改的地方不止上述提到的函数。同时，Value::compare(…)函数也需要修改，以支持date类型之间的比较。具体的实现方式可以查看参考代码。

* + 1. B+树索引的修改

在bplus\_tree.h中，定义了索引中属性比较方法，增加了date字段后需要修改对应的代码，参考其他数据类型的代码即可。

* + 1. 非法输入的处理

测试样例中包含大量的非法输入，如2017-2-29、2017-21-29、2017-12-32等，在实现date字段的基本功能后，需要考虑非法输入的特殊处理。

* 1. 本地测试

Miniob也提供了自动化测试的脚本以及测试用例，测试脚本位于./test/case/miniob\_test.py中，本题目的测试用例位于./test/case/test/primary-date.test中，对应的标准输出位于./test/case/result/primary-date.result中。

可以通过以下命令执行测试：

python3 miniob\_test.py --test-cases=basic

python3 miniob\_test.py --test-cases=primary-date

在执行测试后，测试的执行情况日志保存在 /tmp/miniob/result\_output 中。

* 1. 拓展
* 本指导书采用int类型的数据来存储date，请同学们尝试采用其他类型（比如字符串）或是构造数据类型（结构体）来存储date。
* 同学们观察Value类中定义的已有数据类型（int float char），它们的实现均考虑到了不同数据类型之间的比较情况，比如int类型与float类型在比较时会发生隐式类型转换。请同学们尝试考虑date类型与其他类型是否可以比较，如果可以比较的话可以尝试实现这部分代码；如果不可以比较的话，是否要进行相应的错误处理？