摘要

本实验是数据库系统实现的实践部分，即利用一个现成的小型数据库框架—Miniob，通过向其中添加代码，逐步完善一个数据库应该具有的各种功能，最终实现一个较为完整的数据库系统。

实验过程中，在Miniob原本的一些功能，即Basic部分，的基础上，根据相关测试样例和课程要求的指示，我主要集中完成了Main部分，即Drop-table，Date，Update，Select-table，Show-index，Unique，Expression，Null，Order-by，Group-by， Simple-sub-query等功能要求，同时又因为相关样例的特殊性，除此之外，还加上了Pre-requires部分，即Typecast，Insert，Like，AggrFuncExpr，Having等功能要求。（剩下的，如Multi-index等功能并没有实现。）

在该实验报告中，首先会就Basic部分进行分析，并提供一个整体的宏观思路，随后便是各种功能的具体实现，以及相应的具体思路和前置准备等内容。

**关键词：数据库系统，Miniob，Basic，Main，Pre-requires，相关功能实现；**

目录

[摘要 3](#_Toc156095252)

[1 The Basic 5](#_Toc156095253)

[1.1 前置说明 5](#_Toc156095254)

[1.2 基本框架 6](#_Toc156095255)

[1.4 初步思路 7](#_Toc156095256)

[1.3 宏观说明 7](#_Toc156095257)

[2 The Drop-table 8](#_Toc156095258)

[2.1 修改思路 8](#_Toc156095259)

[2.2 相关文件与实现 8](#_Toc156095260)

[3 The Date 9](#_Toc156095261)

[3.1 修改思路 9](#_Toc156095262)

[3.2 相关文件和实现 9](#_Toc156095263)

[3.3 附加功能的实现（Typecast） 10](#_Toc156095264)

[4 The Update 11](#_Toc156095265)

[4.1 修改思路 11](#_Toc156095266)

[4.2 相关文件和实现 11](#_Toc156095267)

[5 The Select-table 12](#_Toc156095268)

[5.1 修改思路 12](#_Toc156095269)

[5.2 相关文件和实现 12](#_Toc156095270)

[6 The Show-index 14](#_Toc156095271)

[6.1 修改思路 14](#_Toc156095272)

[6.2 相关文件和实现 14](#_Toc156095273)

[7 The Unique 15](#_Toc156095274)

[7.1 修改思路 15](#_Toc156095275)

[7.2 相关文件和实现 15](#_Toc156095276)

[8 The Expression 17](#_Toc156095277)

[8.1 修改思路 17](#_Toc156095278)

[8.2 相关文件和实现 17](#_Toc156095279)

[9 The Null 19](#_Toc156095280)

[9.1 修改思路 19](#_Toc156095281)

[9.2 相关文件和实现 19](#_Toc156095282)

[10 The Order-by 21](#_Toc156095283)

[10.1 修改思路 21](#_Toc156095284)

[10.2 相关文件和实现 21](#_Toc156095285)

[11 The Group-by 23](#_Toc156095286)

[11.1 修改思路 23](#_Toc156095287)

[11.2 相关文件和实现 23](#_Toc156095288)

[12 The Simple-sub-query 25](#_Toc156095289)

[12.1 修改思路 25](#_Toc156095290)

[12.2 相关文件和实现 25](#_Toc156095291)

[13 总结和展望 28](#_Toc156095292)

# 1 The Basic

## 1.1 前置说明

在该实验中，我主要集中完成了Main部分，即Drop-table，Date，Update，Select-table，Show-index，Unique，Expression，Null，Order-by，Group-by， Simple-sub-query等功能要求，同时又因为相关样例的特殊性，除此之外，还加上了Pre-requires部分，即Typecast，Insert，Like，AggrFuncExpr，Having等功能要求。（剩下的，如Multi-index等功能并没有实现。）

相关提测结果的说明，由于官方提测平台的各测试样例是逐步递进的，但一些功能并没有在课程的要求中，故没有实现，比如multi-index等（但即便如此，还是多实现了一些使用频繁的功能，如Typecast，AggrFuncExpr等），所以直接在miniob-2022部分提测，某些依赖该功能的样例并不能通过，但是在miniob部分却可以，比如unique，null。

（Basic，Select-meta部分为系统预实现的，其余的部分可以取Miniob-2022和Miniob的并集。）

1. Miniob-2022通过的部分，Drop-table，Date，Update，Select-tables，Show-index，Expression，Order-by，Group-by(and Having)，Simple-sub-query。（以及Typecast，Insert，Like，Aggregation-func）（下图为miniob-2022的提测结果。）



2. Miniob通过的部分，Unique，Null。（下图为miniob的提测结果。）



## 1.2 基本框架

此部分将会就Miniob本身的基础部分进行解析，重点关注基础功能的实现，在这个过程中对其基本框架进行熟悉，初步了解该小型数据库本身。

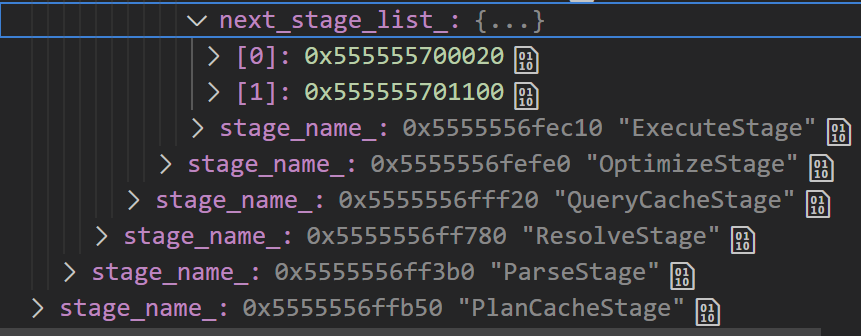
首先是词法和语法分析的部分，涉及src/observer/sql/parser/lex\_sql.l和src/observer/sql/parser/yacc\_sql.y两个文件。Lex\_sql.l涉及相关词法分析，将输入的statement划分为各种token，然后经由yacc\_sql.y进行语法分析，对相应的变量和数据结构进行赋值和初始化。相关初始化函数经由parser\_defs.h和parser.h连接，并在parser.cpp中进行实现（比如value\_init\_integer，create\_index\_init等）。然后就是运行流程。

首先在src/observer/main.cpp中进行了服务器的初始化，然后通过客户端进行连接，并输入语句，此时进行相应的事务处理。

src/observer/session/session\_stage.cpp进行相应的事务处理，其中SessionStage::handle\_event()->SessionStage::handle\_request()为相应的函数以及调用关系。其中参数均为StageEvent（将其强制转换为SessionEvent）。

随后依据此生成相应的SQLStageEvent(该结构体含有四个私有变量，SessionEvent指代该事件，String::sql\_指代语句字符串，Query指代后续生成的语法树， Stmt指代后续的内部数据结构)，然后按照下图的顺序进行各个阶段的处理。（这里交由PlanCacheStage开始顺序处理。）

（顺序如图所示，由下至上）（重点关注其中三个阶段）



1.SQL解析：将用户输入的SQL语句解析成语法树；

ParseStage部分(src/observer/sql/parser/parse\_stage.cpp)：

ParseStage::handle\_event() and ParseStage::handle\_request()。其中handle\_request将会调用parse()函数，对SQLStageEvent中的Query进行相应的赋值（该结构体变量相关的成员见parse\_defs.h，主要是由语法解析出来的关键信息）。然后转向下一个阶段。

2.语义解析：将生成的语法树，转换成数据库内部数据结构；

ResolveStage部分(src/observer/sql/parser/resolve\_stage.cpp):

ResolveStage::handle\_event()。该函数会调用create\_stmt函数（该函数见src/observer/sql/stmt/stmt.cpp，涉及InsertStmt,DeletStmt,SelectStmt的create），对SQLStageEvent中的Stmt进行相应的赋值（）。这些Stmt一般包含，表信息，属性名，过滤条件FilterStmt（包含并列关系的FilterUnit），然后转向下一个阶段。

3.计划执行：根据语法树描述，执行并生成结果；

ExecuteStage部分(src/observer/sql/executor/execute\_stage.cpp):

ExecuteStage::handle\_event() and ExecuteStage::handle\_request()。其中handle\_request依据stmt->type()进行不同的处理，调用如do\_select等实际上的功能函数（其中注意Operator部分，指向查询计划中对应的包含一个或多个子操作符的操作符，使其共同工作以完成整个查询的任务：先扫描每一行，构建PredicateOperator谓词操作符，构建ProjectOperator投影操作符，然后打开并执行操作符，迭代并获取结果。每次迭代都获取下一个满足条件的行，然后进行迭代投影）。（操作符的父子关系：Projection->Pred->Scan）

（后续将要实现的功能，基本上都只是在这些阶段的基础上进行更改补充）

## 1.4 初步思路

在熟悉Miniob的整体结构和运行逻辑的情况下，对于各种待实现的功能，首先进行解析，需要添加，修改什么内容，进而再在代码中定位到相关的部分，在此基础上，修改并完善补充相应的内容，最后调整其他与之相关的部分即可。若出现错误，则利用简单的样例进行调试定位。

## 1.3 宏观说明

在我的实现方案中，我的路径如下：

Basic and Select-meta -> Drop-table -> Date -> Update -> (Typecast) -> Select-table -> (Pre-requires) -> Show-index -> Unique -> Expression -> Null -> Order-by -> (AggrFuncExpr) -> Group-by -> (Having) -> Simple-sub-query.

在后续的各个功能的实现中，我将指出所涉及的源文件（契合整体结构），宏观思路，根据功能逻辑进行的大致修改（契合相关运行逻辑）,以及一些前置要求或Bug修正。（其中括号里前后关联不大的部分没有出现在本报告中）

# 2 The Drop-table

## 2.1 修改思路

首先在基础框架中已经给出了Drop的词法和语法分析，因此应该重点关注执行阶段的处理。而在该阶段的相关文件中，我们可以根据其他功能的实现形式进行模仿，实现do\_drop\_table函数，又由于该操作会涉及数据库对表的具体操作，因此我们还应该在Db类中具体实现相关的操作逻辑；随后进行对Table的删除操作，故在此类中应该实现具体的remove函数。（除此之外，还应该在事务中添加对delete的支持，即修改trx.cpp文件）

## 2.2 相关文件与实现

修改与参考文件如下：

src/observer/sql/executor/execute\_stage.cpp->handle\_request/do\_drop\_table

src/observer/storage/common/db.cpp -> drop\_table

src/observer/storage/common/table.cpp -> remove

src/observer/storage/trx/trx.cpp -> delete\_table

因此实现ExecuteStage::do\_drop\_table,Db::drop\_table，Table::remove等三个函数，并完善handle\_request部分。

ExecuteStage::do\_drop\_table中首先提取相关对象和信息，然后在已经连接的db中查找表，然后在事务中以及db中删除该表，并设置响应。

Db::drop\_table首先检查表名是否存在于打开的表集合中，然后释放表对象的内存，即调用remove函数，最后从表集合中删除该表。

Table::remove，负责移除相关的索引文件，关闭数据文件和清理相关资源，并移除数据文件和元数据文件。

# 3 The Date

## 3.1 修改思路

要实现date类型的字段，我们就必须考虑其解析与存储，所以首先要完成其词法与语法的分析，实现其初始化函数（在这里完成对日期合法的判断）。对于它的存储，我们可以直接复用已经实现的int，然后只是用属性名进行标识，因此只需要在相关文件中添加对DATE类型的索引即可（复用对int的操作）。

除此之外，我们还需要考虑当他出现在过滤条件中的情况，所以我们需要完善FilterStmt（并列的过滤条件FilterUnit的集合），在这里可以临时补充对类型的判断（后续可能更改）。（但这里其实仍没有涉及执行，下一步才会涉及Date的比较）

最后就是有关Date类型的比较过滤，注意Select中的谓词操作符Predicator,用来处理where，其中do\_predicate进行相应的comp处理，这里需要注意TupleCell的作用，用来辅助依赖各个元组（或许已经经过一定的操作）的操作，这里我们需要完善其本身的一些功能函数，TupleCell::to\_string（输出字符串）和TupleCell::compare（值的比较）,在其中添加对Date的支持。

## 3.2 相关文件和实现

src/observer/sql/parser/lex\_sql.l

src/observer/sql/parser/yacc\_sql.y

src/observer/sql/parser/parse.cpp

src/observer/sql/parser/parse\_defs.h

在这三个文件中完成对date的词法以及语法分析，并在parse.cpp中完成date的初始化函数（语法分析时会调用），以及在赋值时对Date进行合法性判断。

src/observer/sql/expr/tuple\_cell.cpp

src/observer/sql/stmt/filter\_stmt.cpp（Undo Finally）

在这两个文件中完成对类型匹配的初步实现（但后续其实删除了这部分，在其他地方实现），以及在对表进行实际过滤操作时所依赖的相关功能函数的完善（将值转换为字符串to\_string，以及where筛选条件左右表达式取值赋给TupleCell后的比较函数compare）。

src/observer/storage/common/condition\_filter.cpp

src/observer/storage/common/field\_meta.cpp

src/observer/storage/default/default\_storage\_stage.cpp

src/observer/storage/index/bplus\_tree.h

在这几个文件中添加对Date的处理（作为 case DATES:），可以直接复用Int的功能部分。

## 3.3 附加功能的实现（Typecast）

src/observer/util/typecast.cpp，在typecast.cpp中维护一个二维类型转换函数指针数组cast\_to(存储各个具体的类型转换函数),用来完成相对应的类型转换，修改table.cpp中的make\_record函数,使其可以使用typecast中的函数数组实现类型转换（因此还需要修insert\_stmt的create函数，将其中check fields type部分删除掉）。最后在TupleCell::compare函数中增加对类型转换的支持，先进行判断，再进行转换比较（在我实现的代码中，先是把类型转换放在了do\_predicate,该函数对过滤条件FilterUnit比较符左右的表达式进行取值，设置好TupleCell后进行比较。后面发现类型转换的内容其实应该隶属于compare，故进行了Code Adjustment）。

# 4 The Update

## 4.1 修改思路

要实现Update功能，由于词法与语法分析上已经支持，但是在语义解析上缺少相应的Stmt的数据结构，所以我们需要仿照InsertStmt，DeleteStmt等实现UpdateStmt类，并实现相关函数，如create（stmt.cpp中语义解析部分直接调用的函数）。

然后就是ExecuteStage执行阶段，我们需要在handle\_request中添加对新实现的do\_update的应用，并且注意到等待实现的Operator（用来进行实际的功能操作，如Select依赖ProjectOperator，PredicateOperator等），因此我们需要完善update\_operator.cpp（主要是其中的三个函数open/next/close）。

最后便是对实际存储的数据进行更新（于UpdateOperator::open中调用），因此还需要实现update\_record函数。

## 4.2 相关文件和实现

src/observer/sql/stmt/stmt.cpp -> create\_stmt

src/observer/sql/stmt/update\_stmt.cpp -> create

在stmt.cpp中的create\_stmt中引用UpdateStmt::create，并实现该create函数，主要是获取表的相关信息，并进行检查，创建相应的过滤语句，从而添加对Update的语义解析，

src/observer/sql/executor/execute\_stage.cpp -> do\_update

src/observer/sql/operator/update\_operator.cpp -> open/next/close

在ExecuteStage::do\_update中获取语句，数据库，事务等相关对象，然后初始化表扫描操作符TableScanOperator，PredicateOperator以及新实现的UpdateOperator，然后开始迭代操作。UpdateOperator负责对每一个符合要求的元组进行更新。至于open/next/close三个函数，其中next无需实现（毕竟UpdateOperator只需要对当前的元组完成更新即可），open先打开相应的子操作符（上面的那两个），在其结果集上进行循环遍历，使用下面的update\_record更新表中的记录。

src/observer/storage/common/table.cpp -> update\_record

src/observer/storage/trx/trx.cpp -> update\_record

在对元组更新的过程中，我们还需要对相应的数据库中的每条记录（即元组）进行实际上的更新操作，因此需要在table.cpp中实现update\_record函数：先是初始化变量，用于记录属性的偏移，长度等信息，遍历字段，找到对应的属性，进行类型匹配的检查，然后备份旧记录数据，为新记录添加索引项，最后更新记录数据，删除旧的索引项。最后还可以在update\_record中为事务创建记录，添加上对事务的支持，以期完成clog。

# 5 The Select-tables

## 5.1 修改思路

首先，该功能对原语句提出了进一步的要求，因此还是需要先进行词法和语法的解析，使其可以识别出多张表。随后进入下一个主要阶段，语义解析部分也要求对SelectStmt进行相应的功能扩充（但实际上只是做了同义更替）。又因为该功能要求对多张表进行笛卡尔积关联，因此我们需要实现一个JoinOperator，来生成相对应的复合元组，然后再进行相应的操作，如PredicateOperator，ProjectOperator，故它们也需要进行相应的调整。对于该复合元组，我们需要扩展原本的数据结构，利用一个新的类CompoundTuple来进行存储，因此在sql/expr/tuple.h中增加了该类。最后由于数据结构的改变，一些与之相关的函数也需要进行调整，比如参数的类型等等。（以及在后续实现过程中发现，还需要借用一个Record的vector，即CompoundRecord）

## 5.2 相关文件和实现

src/observer/sql/parser/lex\_sql.l

src/observer/sql/parser/yacc\_sql.y

这两个文件涉及相应的词法（主要是增加了!=）和语法解析。

src/observer/sql/stmt/select\_stmt.cpp （no use）

把循环遍历多张表的部分进行了同义更替

src/observer/expr/tuple.h

其中包含新定义的CompoundTuple类，继承Tuple，同时有左表和右表指针，可以在逻辑上实现两表之间元组的连接。（迭代之中可以实现多表的元组连接，至于查询可以依据index和index-num在左右进行查找）

src/observer/executor/execute\_stage.cpp

在该文件的do\_select中新增对多表的支持，即，如果SelectStmt中的tables大于1，则先进行笛卡尔关联do\_join。在do\_join中，先是创建操作符存储列表，然后获取查询中的表和过滤条件，并对过滤条件进行分割split\_filters（将相应的过滤条件加入到相应的表的过滤条件列表中），然后对遍历每个表，创建相应的扫描操作符，放入存储列表中，然后利用JoinOperator依次进行连接操作，直到存储列表中只有一个操作符，然后返回该操作符（即完成了笛卡尔连接）。随后在do\_select中按照原本逻辑进行迭代操作（创建Predicat，创建Project）。

src/observer/operator/join\_operator.cpp

src/observer/operator/predicate\_operator.cpp

src/observer/operator/project\_operator.cpp

PredicateOperator和ProjectOperator（对相应的参数进行修改，同时ProjectOperator中对于多表查询而言，展示的alias需要带有表名字，利用is\_single\_table进行标识是否是多表）。至于新实现的JoinOperator，open利用新定义的CompoundTuple存储每一次操作生成的复合元组（即用其中的左表和右表指针分别指向两表之间的各个元组，实现逻辑上的连接），next则不断选取右表中剩下的元组进行组合（到底后再移动左表指针），然后再对此时已更改的CompoundTuple进行后续操作。如此便实现了多表连接时的复合元组的生成（逻辑上）与处理。（此时还需要注意一点，倘若是多表连接，后续CompoundTuple中的左右表指针便不仅仅只是指向一个Record，而是多个已经复合后的Record，即CompoundRecord）

src/observer/storage/record/record.h

其中定义了CompoundRecord，为record的vector数组。

# 6 The Show-index

## 6.1 修改思路

首先还是先实现相应的词法和语法分析，然后考虑到现有的结构，我们主要是在ExecuteStage部分进行补充，定位到handle\_request中的SCF\_SHOW\_INDEX,我们在这里调用do\_show\_index，完成逻辑上的功能（调用后续的show\_index），最后由于需要展示实际的属性内容（隶属于TableMeta中），所以我们需要在table\_meta.cpp中完成实际上的show\_index函数，先是打印出header，然后循环打印出该表中存在的index（调用index\_meta中的show函数）即可。

## 6.2 相关文件和实现

src/observer/sql/parser/yacc\_sql.y

该文件中实现show index的语法解析。

src/observer/sql/executor/execute\_stage.cpp

完善该文件中的handle\_request函数，为show\_index增加功能函数，ExecuteStage::do\_show\_index，先是查询对象和数据库，然后获取表名进行查找，展示其索引信息，即调用TableMeta::show\_index.

src/observer/storage/common/index\_meta.cpp

src/observer/storage/common/table\_meta.cpp

这两个文件实现相对应的打印函数，TableMeta::show\_index打印出合适的Header，然后调用IndexMeta::show分别打印出各个index。

# 7 The Unique

## 7.1 修改思路

首先还是相应的词法，语法分析（针对Unique），然后又因为该Unique基本与index合用，因此可以考虑复用原本index的相关内容（即上一个实现的do\_create\_index等部分），在此基础上在原本的结构体CreateIndex中添加一个名为unique的bool变量进行标识即可，随后将所有index涉及的函数有选择性地添加一个unique参数（从初始化函数到后续的功能调用）。除此之外，当对index进行存储的时候，我们还需要保留该unique的信息，所以在index\_meta.cpp中我们需要对to\_json进行完善，使之能够保持unique信息。

除此之外，在本部分中，我还调整了TableMeta和IndexMeta的打印函数，功能没有太大变化。

## 7.2 相关文件和实现

src/observer/sql/parser/lex\_sql.l

src/observer/sql/parser/yacc\_sql.y

src/observer/sql/parser/parse.cpp

src/observer/sql/parser/parse\_defs.h

为unique提供词法和语法支持，为CreateIndex新增一个unique变量，同时将其所涉及的所有相关函数增加一个参数unique。

src/observer/sql/executor/execute\_stage.cpp

src/observer/storage/common/index\_meta.cpp

src/observer/storage/common/table.cpp

src/ovserver/storage/common/table\_meta.cpp

src/observer/storage/default/default\_handler.cpp

src/observer/storage/index/bplus\_tree.cpp

src/observer/storage/index/bplus\_tree\_index.cpp

src/observer/storage/index/bplus\_tree.h

在以上的文件中，均是对unique变量的适应性调整。

以上完善了一下show index的逻辑,并且新增了对unique的词法和语法支持,同时在一些必要的数据结构中添加上unique字段进行标识,并新增一些辅助函数,更改一些函数的参数。（但是存在bug，即在插入时，还是会把相同的unique属性的元组插入进去，并没有报出重复关键字的错误）

因此我通过追踪RECORD\_DUPLICATE\_KEY的相关位置，发现

src/observer/storage/index/bplus\_tree.cpp ->

insert\_entry\_into\_leaf\_node() ->

LeafIndexNodeHandler:leaf\_node.lookup() ->

class KeyComparator::operator()这一比较逻辑线路，可知如果v1=v2,则result=0,因此会返回其他比较函数的值,故因此这里基本没有duplicate的错误,但是如果该值为unique,则需要直接返回相等,触发duplicate的异常,所以在这里添加上unique\_的判断比较(若为零,则直接返回result，0的值,如果result此时不为零则寻得一个插入位置,否则触发duplicate异常).

为了契合上面的实现，我们需要为index提供unique支持,所以BplusTreeIndex::create以及IndexMeta::init,KeyComparator::init中我们需要添加一个unique的参数.

# 8 The Expression

## 8.1 修改思路

通过分析本功能的要求可知，同之前已经实现的内容相比，差别在于原本是对于属性列进行相关操作，而现在则是把属性列（原本用query\_fields,即有关Field的vector）按要求扩展为较为复杂的表达式，因此首先需要更改原词法和语法的分析部分，采用另一种方式替代存储此表达式。这也会导致Stmt阶段相应的filter\_stmt和select\_stmt的生成也要进行更改。

后续还存在一些问题，一个是ExecuteStage时相关表达式如何get\_value（这里要求实现与复杂表达式相契合的get\_value函数，依赖一个新的类BinaryExpression）；另一个，则是在投影的时候，如何生成与表达式相契合的投影列名。（可理解为from前后出现的表达式的处理）

## 8.2 相关文件和实现

src/observer/sql/parser/lex\_sql.l

src/observer/sql/parser/yacc\_sql.y

src/observer/sql/parser/parse\_defs.h

src/observer/sql/parser/parse.cpp

上面四个文件涉及和功能需求相关的词法以及语法分析上的更改（同时新增ADD，SUB，DIV，MUL等新识别的token）。为了契合Expression的功能，我选择在parse\_defs.h中引入四个新的结构体UnaryExpr（也可用来指代单一属性或者投影列中的单项表达式），Expr，BinaryExpr（可用于where后面的筛选表达式）（前三个类互相指代，递归定义的情况下可以实现多重表达式）以及ProjectCol（用来指代select语句中等待投影的列，可以包含表达式，后续依赖gen\_project\_name即可实现投影列中的表达式的显示），随后在parse.cpp中完成相应的初始化函数以及功能函数的设置，如projectcol\_init\_star,projectcol\_init\_expr等。此时再更改Condition结构体，现在它只包含三个成员，Compop，左Expr，右Expr，后续可以对左右Expr进行迭代求值。

src/observer/sql/stmt/filter\_stmt.cpp

src/observer/sql/stmt/select\_stmt.cpp

首先是对于filter\_stmt，这里由于上面对condition的成员做出了改变，因此，在语义解析部分，create\_filter\_unit即创建各个并列的筛选条件的时候，就不能继续沿用之前的逻辑，故此处引入create\_expression对Condition左右Expr进行求值，要么是ValueExpr，要么是FieldExpr（各自有不同的get\_value逻辑）。

然后就是select\_stmt，对于投影列中所存在的表达式，在ParseStage阶段，我们采用ProjectCol存储，此时要进行语义解析，因此这里我们引入一个函数gen\_project\_expression（通过递归的结构实现更复杂的表达式）用以生成与ProjectCol相对应的数据结构。

src/observer/sql/executor/execute\_stage.cpp

src/observer/sql/operator/project\_operator.cpp

src/observer/sql/expr/expression.h

src/observer/sql/expr/tuple\_cell.cpp

然后就是ExecuteStage阶段，此处我们要根据投影列的变化（引入了ProjectCol）对原本的do\_select中投影操作进行适配。同时在其所引用的add\_projection中也要对相应的投影列名进行规范化，这里我们引入了gen\_project\_name（将投影列的表达式转换为字符串，然后set\_alias），然后print\_tuple\_header。(至于接下来的每一行，则通过设定好的Operator进行迭代处理即可)。然后就是有关表达式具体求值的核心部分（即如何根据语义解析出来的递归形式的表达式，满足其求值的需求），也就是get\_value的实现。

在程序的本来框架中存在两个get\_value，FieldExpr::get\_value和ValueExpr::get\_value。后者对应语义解析后常量部分，前者对应语义解析后元组中特定的一部分（在运行过程中不断在RowTuple中find\_cell，然后返回该位置的确切值）。但是对于表达式，我们需要进行进一步的运算，于是这两个get\_value就不适用了。考虑到运算表达式的性质，即基本都包含二元操作符，因此可以新创建一个类BinaryExpress，用以表示包含二元操作符的表达式，其中它的get\_value采用递归处理（不断求值二元运算符两端的表达式），最终可以实现基于该表达式，对每一个复合元组的正确求值。

其中tuple\_cell.cpp涉及四个二元操作符的具体实现sub，mul，div，add。（其中可以进行typecast。）

src/observer/storage/common/condition\_filter.cpp

根据Parser阶段引入的新结构体，以及相对应的函数，可以对其中的内容进行同义更新。

但这里仍然存在一个问题，即如果表达式中存在1/0这种情况时该如何处理。应该将这种没有定义的表达式的值设置为Null，然后才算是补全了这一部分的内容。（故此处依赖下面的Null的实现）

# 9 The Null

## 9.1 修改思路

首先，我们可以通过在属性值类型中增加NULLS来指代当前值为null的情况，同时为属性信息增加nullable来判断其是否可以为null。然后就是对词法分析和语法分析部分进行相对应的改变，同时处理由此带来的对其他部分的影响。

同时为了保持nullable的信息，我们需要修改field\_meta和table\_meta的部分，同时为了方便我们后续在使用过程中判断某一项属性是否nullable，我们选择增加额外的一个字段来表示bitmap，用bit位来指示前面的属性哪些可以为null。正因为如此，在其他地方我们需要考虑这新的一列所带来的影响，比如在table.cpp中就需要进行相应的更改。

## 9.2 相关文件和实现

src/observer/sql/parser/lex\_sql.l

src/observer/sql/parser/yacc\_sql.y

src/observer/sql/parser/parse.cpp

src/observer/sql/parser/parse\_defs.h

增加了对null，nullable的词法支持，然后便是相对应的一些语法支持，对于每一个属性（即AttrInfo）我们增加nullable变量，我们默认nullalbe为false，因此需要对相应部分进行适应性更改（比如attr\_info\_init）。以及在属性值类型AttrType部分增加一个NULLS，用以指示该值是否为null。

src/observer/sql/expr/tuple.h

src/observer/sql/expr/tuple\_cell.cpp

src/observer/sql/expr/tuple\_cell.h

在tuple\_cell的相关部分，即sub，mul，div，add中添加对null的判断，如果其中一个TupleCell为null，那么直接返回null，同时在div中注意增加对除以0的判断，以及null的to\_string部分。除此之外，还实现了一些功能函数，如is\_null()，set\_null()。

src/observer/sql/operator/predicate\_operator.cpp

src/observer/sql/operator/table\_scan\_operator.cpp

在do\_predicate中，注意实现题目的要求，比较过程中，如果一方为null，那么直接返回false。

至于table\_scan\_operator.cpp中涉及一个比较奇特的bug，官方已经解决。

src/observer/sql/stmt/insert\_stmt.cpp

src/observer/sql/stmt/select\_stmt.cpp

这两个文件注意与其他部分的更改保持同步就行（比如在相关操作中考虑下面新增的那一列）。除此之外，在insert\_stmt中我们可以在create中初步实现字段类型的null检查。

src/observer/storage/common/field\_meta.cpp

src/observer/storage/common/table.cpp

src/observer/storage/common/table\_meta.cpp

由于属性部分新增了nullable，故这些部分需要进行相适应的更改，比如参数的增加等等。同时我们在属性列后面增加一列，储存bitmap（相对应的RowTuple中直接可以使用），用以指示前述的属性是否可以为null，这样可以通过对bitmap的查询迅速得知其是否可以为null，方便进行判断。（在table\_meta中可用extra\_filed\_num来表示这额外添加的一项，故当进行相关操作的时候，注意是否需要考虑这新增加的一列）

src/observer/util/typecast.cpp

这里新增其他类型与null之间的交互。

# 10 The Order-by

## 10.1 修改思路

首先是其词法和语法上的分析（毕竟原框架没有实现），我们需要利用一个Orderby结构体，用来指示相关的属性，以及升序or降序，然后完成相对应的初始化以及操作函数

然后就是语义解析部分，又由于该功能经常和select合作，因此考虑修改SelectStmt的生成，在其中添加OrderByStmt的生成（用SelectStmt中的一个成员指代即可）（具体的实现见OrderByStmt，和FilterStmt一致，采用units的办法分别存储并列的排序要求）

ExecuteStage阶段，我们需要在do\_select的操作符部分引入一个新的操作符，即SortOperator（位于ProjectOperator之前，PredicateOperator之后），实现预期的功能。而这里则要求对TupleCel进行比较，因此需要在tuple\_cell.h中实现相应的比较函数。至于实际的Operator（见相应的文件中）。

## 10.2 相关文件和实现

src/observer/sql/parser/lex\_sql.l

src/observer/sql/parser/yacc\_sql.y

引入ASC，ORDER，BY的词法分析，并且仿照condition部分修改实现orderby的语法分析部分（经过上面的分析可知，在执行阶段之前，orderby的要求和filter的要求有些类类似，故可以参考实现）。

src/observer/sql/stmt/filter\_stmt.h

src/observer/sql/stmt/select\_stmt.h

src/observer/sql/stmt/select\_stmt.cpp

src/observer/sql/stmt/orderby\_stmt.h

src/observer/sql/stmt/orderby\_stmt.cpp

src/observer/sql/expr/tuple\_cell.h

然后就是实现相应的OrderByStmt，其中注意功能函数create和create\_orderby\_unit，用以进行对orderby的语义解析。最后该功能一般和select连用，因此，我们选择为SelectStmt添加一个成员OrderByStmt（和FilterStmt一样），同时在其create函数中增加对orderby的生成引用。

src/observer/sql/operator/sort\_operator.h

src/observer/sql/operator/sort\_operator.cpp

这些文件为SortOperator的核心实现逻辑，首先通过fetch\_and\_sort\_table函数从输入的子操作符中获取记录并进行排序，然后在next函数中按书顺序填充已排序的记录。（和其他操作的配合可以结合Select-table部分分析）

Fetch\_and\_sort\_table首先循环遍历子操作符的记录，不断构建获得各元组单元，并构建一对记录，加入排序表向量，然后使用排序函数对排序表进行排序，随后填充有序索引表，并设置迭代器初始位置。最后在next中不断填充返回该有序索引表中的每一项。

src/observer/sql/executor/execute\_stage.cpp

这个文件主要是在do\_select中添加SortOperator操作符。

# 11 The Group-by

## 11.1 修改思路

（由于测试样例的特殊性，这里还要去实现前置的AggrFunc。故下面的思路是在此基础之上的。但是在我的解决方案中，其实AggrFunc也只是提前完成了一部分，后续还是需要依赖Group-by的内容才算完整。）

首先还是词法以及语法分析，（可以参考OrderBy进行补充，我认为包括Filter在内的三者有极高的相似性）。然后就是语义解析阶段，需要实现GroupByStmt，同时这里还需要考虑到一点，究竟如何实现GroupBy操作，我采取的方式是先按照GroupBy后面的分组标准对原表进行排序，如此相同分组标准的元组便处在相邻的位置，此时再对排序后的结果进行有条件的顺序扫描即可划分出所有分组。在实际操作时会借用GroupByOperator，（结合前述实现的Operator），其中我们可以将聚合操作分为两种，一个是同一分组之间的do\_aggregate，另一个是不同分组之间的do\_aggregate\_first，这样在该操作符中即可完成相应的聚合操作。除此之外，结合Select-table ,考虑到聚合时也会产生新的元组类，故可以选择实现GroupTuple进行相关的参数传递。

最后便是完善do\_select的操作符那一部分，将OrderBy要求的逻辑融入进去即可。

## 11.2 相关文件和实现

src/observer/sql/parser/lex\_sql.l

src/observer/sql/parser/yacc\_sql.y

src/observer/sql/parser/parse.cpp

src/observer/sql/parser/parse\_defs.h

首先实现Group的词法分析，然后仿照orderby的相关内容对Group的语法分析进行补充，在填上一些辅助信息，如groupby\_length和groupbys[MAX\_NUM]（其中包含groupby的属性名以及相应的表名，后续依赖此进行相关操作）。在parse\_defs.h中我们在Selects的结构体中和orderby一样添加了groupby的一些数据（即上面两个），以及在parse.cpp相应的初始化函数selects\_append\_groupbys。

src/observer/sql/stmt/groupby\_stmt.h

src/observer/sql/stmt/groupby\_stmt.cpp

src/observer/sql/stmt/orderby\_stmt.h

src/observer/sql/stmt/orderby\_stmt.cpp

src/observer/sql/stmt/select\_stmt.h

src/observer/sql/stmt/select\_stmt.cpp

在语义解析部分，首先和orderby一样，groupby与select关系很密切（从上面添加的辅助信息也可看出），所以首先要在SelectStmt里添加对GroupByStmt的指代，同时我们还需要一个OrderBystmt来指代orderby\_stmt\_for\_groupby（先通过这一项排序，才能将一个或者多个列按照分组标准进行排序，后续才可以直接通过GroupBy顺序处理，直接将相同分组标准的合并，因为它们此时是相邻的）。又正因为如此，我们需要在OrderByStmt中额外实现以GroupBy为参数的create和create\_orderby，用以生成orderby\_stmt\_for\_groupby。最后便是GroupByStmt以及GroupByUnit的相关实现，同FilterStmt以及OrderByStmt本质上差别不大。（因此可以参考OrderByStmt进行语义解析）

src/observer/sql/expr/tuple.h

src/observer/sql/expr/tuple.cpp

在Select-table中，由于出现了笛卡尔积，出现了一种新的元组即CompoundTuple，通过类比，我们不难想到对于那些会产生新的元组的操作，往往会借用一个新的元组类型。因此，这里我们选择实现一个新的元组类，即GroupTuple，用来存储完成排序后，同一分组合并时的元组。其中除了Tuple类的相关方法，应该注意到，groupby语句通常用于配合聚合函数，根据一个或多个列对结果集进行分组，一般下情况，需要利用聚合函数对组内结果进行计数，求和等操作。所以这里我们实现了三个新函数，GroupTuple::do\_aggregate\_first，GroupTuple::do\_aggregate，GroupTuple::do\_aggregate\_done，用以进行同一分组下的聚合函数操作。（这些函数会在GroupByOperator的next中被调用，以期获得正确的操作）。

src/observer/sql/executor/execute\_stage.cpp

该文件中主要是完善do\_select函数的逻辑（将GroupBy添加进去）。因此整体而言，相关操作符的顺序为JoinOperator，PredicateOperator，（此处是GroupByOperator的相关部分），（如果有Having的话，位置在这里），SortOperator，ProjectOperator。

有关GroupByOperator部分，需要注意，首先需要为GroupBy生成SortOperator（按照GroupByUnit进行排序，让相同分组的相邻），再获取聚合函数语句，然后对GroupByStmt进行检查，最后依赖上面的结果，继续生成GroupByOperator（该操作符的详细部分见下）。

src/observer/sql/operator/groupby\_operator.h

src/observer/sql/operator/groupby\_operator.cpp

这两个文件定义并实现了GroupByOperator类。重点关注其中next函数。它依赖三个bool变量进行辅助判断，is\_first\_，is\_new\_group，is\_record\_eof\_。如果是新的分组（is\_new\_group），则调用do\_aggregate\_first进行聚合操作，然后调整当前元组是否属于新的分组，如果不是新分组，则执行聚合函数do\_aggregate并更新结果，如此迭代操作，直到记录末尾。（这里还会用到AggrFuncExpression的get\_value，此处不再赘述。）

# 12 The Simple-sub-query

## 12.1 修改思路

首先考虑到子查询往往是作为where后的condition的一部分（即一个查询条件的左or右的单项表达式），所以我们可以对相关结构体进行扩展，使之可以指向该子查询，SubQueryExpr类型只是ExpType中的一种。根据这种思想对相应的语法分析进行补充。

然后就是存在于FilterUnit某一边的子查询语句，我们期望能够为其生成相应的操作符，进而在后续Express的get\_value中可以不断迭代获取相应的值。于是参考以往的思路，此处可以实现SubQueryExpression用来指代相关子查询语句，对于该语句，我们希望能够在其对应的create\_expression中生成子查询的SelectStmt（可以复用直接复用SelectStmt::create）。然后在后续的操作符的生成中，判断如果他是子查询（由一个flag标识），那么就递归调用生成操作符的那段代码（于是这里要求将其抽象为一个功能函数，即gen\_phsical\_plan），以此生成子查询的操作符，后续便可以通过相应的Expression的get\_value函数去调用其操作符的next，不断获得子查询中的元组，在do\_predicate函数（PredicateOperator的核心判断函数）中实现IN和EXSITS的功能（前者从头将子查询的表遍历一遍进行判断，后者则直接看能否返回一个元组即可）。

因此重点关注，语法解析，子查询的Stmt的生成，子查询的Operator的生成，如何对子查询表达式（SubQueryExpression）进行求值（不断返回相应的元组），以及在谓词操作符中实现IN和EXSIST的功能。（这些其实都可以参考递归复用的思路，做出条件分流即可）

## 12.2 相关文件和实现

src/observer/sql/parser/lex\_sql.l

src/observer/sql/parser/yacc\_sql.y

src/observer/sql/parser/parse.cpp

src/observer/sql/parser/parse\_defs.h

首先是词法分析，比较简单，增加IN和EXISTS即可。

然后是语法分析，改动较多，

在parse\_defs.h的CompOp结构体中添加sub\_query相关的in / not in / exists / not exists。然后再把ExpType进行扩展（即Expression的类型，二元表达式，单项表达式，或者聚合函数等，即一个等待取值的部分），添加SUBQUERY（表示这也算是一个求值部分，逻辑上可行）。然后就是SubQueryType的类型，由上面的in / not in / exists / not exists决定，以及一些通用的类型。后续在Expr结构体中增加对SubQueryExpr的指代，该结构体有一个成员即Selects（Select结构体）。然后便是在parse.cpp中完成相应的初始化和功能函数（如sub\_query\_expr\_init）。

注意其中sub\_query\_expr\_init函数，这里将子查询和父查询进行了关联。再结合上面的内容，需要对后面的逻辑进行一定程度的更改。（sub\_select的语法分析是作为where后面condition的一部分存在的，即此刻in/exists后面的单项表达式unary\_expr）。

src/observer/sql/stmt/filter\_stmt.h

src/observer/sql/stmt/filter\_stmt.cpp

src/observer/sql/stmt/select\_stmt.h

src/observer/sql/stmt/select\_stmt.cpp

src/observer/sql/stmt/stmt.cpp

src/observer/sql/stmt/update\_stmt.cpp

将FilterStmt中的Create\_expression分别在不同的Expression类中各自实现，实现良好的功能划分。SelectStmt,UpdateStmt,Stmt.cpp部分进行了一些安全上的补充，以及消除了一些由于参数变化带来的影响。这里算是进行了同义重构。这里需要注意与后面所实现的SubQueryExpression的联动。作为condition中过滤条件IN/EXISTS的一部分——右表达式，本身按照原本的逻辑应该可以进行create\_expression(而这新的一部分恰好由下面实现)。同时对于不同的Expression，应该有其特殊的get\_value。

src/observer/sql/expr/tuple\_cell.h

src/observer/sql/expr/expression.h

src/observer/sql/expr/expression.cpp

在这三个文件中，完成对create\_expression的重构，将原本处在FilterStmt下的create\_expression移至expression.h中，将相对应的create\_expression分别在各个expression类中（如BinaryExpression，ValueExpr，FieldExpr等）实现，进行良好的划分，并修改其他引用这里的代码。

又由于我们引入了SubQueryExpr，因此我们还需要实现SubQueryExpression这个类，并完成其初始化以及功能函数，其中需要注意SubQueryExpression::create\_expression部分。这是简单子查询的语义解析的重要部分，以及SubQueryExpression::get\_value部分。（这两个部分是整个实现逻辑的关键之处）。

我们希望能够在SubQueryExpression::create\_expression的时候就对对子查询进行相应的SelectStmt::create，然后交由后面阶段生成相应的操作符。同时get\_value函数能够不断得到子查询中的元组（不断next，然后借由此在外围进行比较判断）。（这里依赖后续Operator中的相关实现）。

src/observer/sql/executor/execute\_stage.h

src/observer/sql/executor/execute\_stage.cpp

在这两个文件中，我稍微将do\_select中某些部分打包成了一个函数，即相应操作符的使用->gen\_phsical\_plan，在这里实现各个操作符的级联。本质上仍然是同义表达。（仅仅是做出了格式上的优化）。

但除此之外，SubQuery作为Condition中的一部分，我们需要在PredicateOperator时首先处理Condition中所包含的子查询（前面create\_expression对相应子查询生成的SelectStmt），配合后续更改的PredicateOperator完成相应的条件筛选。

我们需要为FilterUnit中子查询所生成的Stmt生成相应的操作符列表，然后进行SubQuery，这里可以采取递归处理的方式，如果是子查询，我们就再调用gen\_physical\_plan为其生成操作符。（可以采用lambda表达式对每个condition都进行一次处理，确保每一个子查询都能得到操作符）。

src/observer/sql/operator/join\_operator.cpp

src/observer/sql/operator/operator.h

src/observer/sql/operator/predicate\_operator.h

src/observer/sql/operator/predicate\_operator.cpp

join\_operator.cpp和operator.h仅仅只是做出了适应性调整，重点在于predicate\_operator。原因在于，谓词操作符本身就需要处理condition的一部分，由于我们将SubQuery扩展进了里面，所以需要进行相应的修改。因此通过追踪PredicateOperator::next的实现，我们得知需要调用do\_predicate对每一个由子操作符生成的复合元组CompoundTuple进行判断。故需要在这里实现IN / NOT IN / EXISTS / NOT EXISTS等谓词逻辑。

同时利用上面得到的子查询操作符，我们可以通过open\_sub\_query，get\_value，close\_sub\_query得到不断得到子查询中的元组（主要是通过get\_value中对相关子查询的操作符的调用next不断得到下一个值）。Exsit即去判断能够取到一个值即可，IN则需要通过一个while循环将其中所有值都拿出来比较。（这里可以使用Lambda表达式进行简化）。

# 13 总结和展望

通过本实验，我在一定程度上完成了对Miniob的功能补全，在逐步递进实现各功能的过程中，对于数据库系统的实现有了进一步的理解。同时在对于源码的阅读补全中，我对相关功能的认识也更加深刻，也体会到了其中的奥妙。虽然其中经历了许多困难，但是解决之后，收获确实不少。

对于本次课程的要求，我完成了除了clog以外其他所有的功能，以及为了通过特定的样例，还额外实现了一些其他的功能，作为Pre-requires在我的项目中存在。

至于那些尚未实现的部分，比如text，join-table这类有一定的实现思路，并且往往只需要在原本已经实现的功能的基础上简单修饰即可，后续可以考虑在假期继续。至于那些比较复杂的功能比如multi-index，clog等等，后续可能需要进一步思考与学习。

在整个功能实现的过程中，除了那些比较平常的改动外，有这么几个点令我印象极为深刻：

1. 对于Expresssion的渐进实现，我接触到了一种方法，即对于哪怕表面上看起来截然不同的式子，比如单项表达式，子查询等等，都可以用Expression来指代，除了相关数据结构的委派，最为核心的则是get\_value函数的实现，这里提供了一个抽象的结构，可以在不关注Expression本身的差别之下，统一实现求值操作(其中往往又是通过递归迭代的方式实现复杂式子的求值)，当然具体的求值方式差别比较大。
2. Operator部分，令我印象深刻的点在于操作符的生成（对于最后gen\_phisical\_plan部分），这里可以利用条件分流的方式直接递归复用生成子查询的操作符。除此之外，还可以通过巧妙地设计Operator，使用其open/next/close函数，在迭代过程中即可实现相应的逻辑操作。在仅使用一个最顶层的Operator的情况下（通常为ProjectOperator），可以迭代处理得出期望的值。（这个实验最终基本完成了操作符列表整体的完善）

在具体的功能实现的过程中，我主要采取对重要节点进行断点的方式然后进行调试，从而把握整体的运行结构，也方便接下来进一步的功能实现。在中途遇到问题时，一般采用逐渐深入的方式设置断点，进而逐步精确定位。在整个实现过程中，由于整体的项目比较大，发生过几次极其细微但是很致命的错误，比如初始化函数中遗漏了某个值，变量赋值时不小心错位，甚至变量名也进行了误用。不过，经过这一次此的试错与定位，我认为自己对于初具规模的项目的掌握能力更近了一步。

当然，我目前基于的是miniob-2022，相比较于2023比较落后，后续可以继续参考目前已经实现的功能，已经近一年来的更新，争取在2023上实现相应的功能。