

Aggregation 的实现

在实现本功能前，需要先掌握以下知识：

- ✧ 添加新的语法以及对新语法进行解析
- ✧ 添加新的数据结构

为了便于，本文档中的符号说明如下表所示：

符号	说明
SELECT	选择语句标识
attr	属性名(列名)，例如 id, name, age 等等
FROM	与 SELECT 搭配使用
rel	关系名(表名)
WHERE	条件语句标识

0. 前置概要

按照一般的语法规则，在 SqlServer、Mysql 中 Aggregation 的语法如下：

SELECT FUNC(attr) FROM rel WHERE ...

其中 FUNC 为聚合函数，比较常见的函数有 SUM、MAX、MIN 以及 AVG 等等，本实验中主要实现提到的这 4 种聚合函数。

在实验文档【】中已经阐述了 miniob 数据库系统的结构以及代码文件中各模块的位置和功能，这里简单回顾一下代码结构，如图 1 所示。

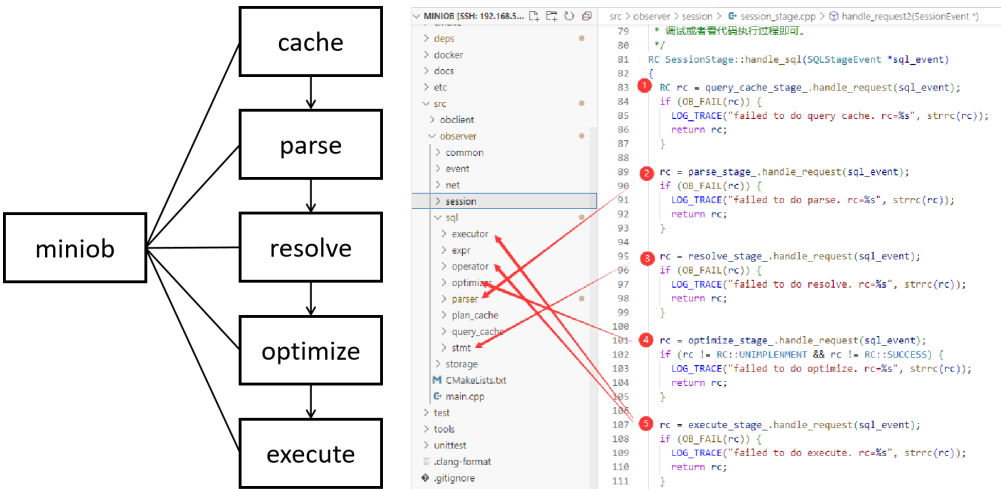
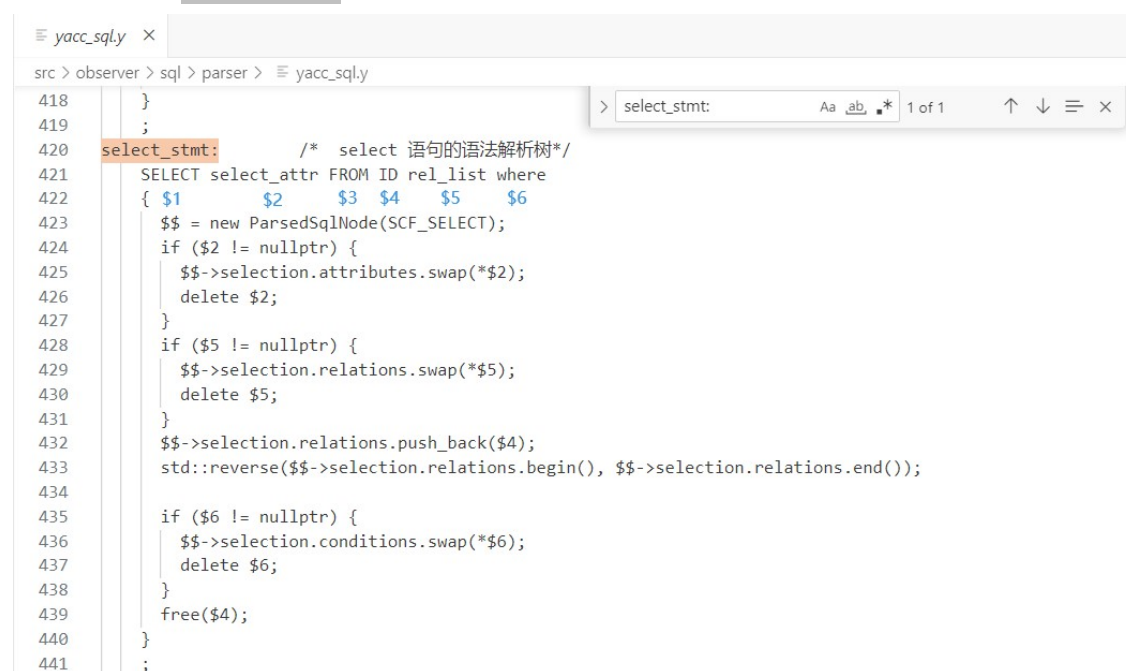


图 1 miniob 代码结构

在后续实验中，主要以 SUM 函数为例进行展开，感兴趣的同学 MAX、MIN、AVG 以及更多其他的函数可以自己动手实现。

0. 语法解析

聚合函数主要配合 SELECT 语句进行使用，因此我们需要在 SELECT 语法规则的基础上进行扩展，增加对聚合函数的识别和支持。在 minioib 中，已经实现了一般 SELECT 语句的语法解析，该部分可以在 ./src/observer/sql/parser/yacc_sql.y 中可以找到（查找 `select_stmt:` 关键词）。



```
418     }
419     ;
420 select_stmt:      /* select 语句的语法解析树*/
421     SELECT select_attr FROM ID rel_list where
422     { $1      $2      $3 $4      $5      $6
423       $$ = new ParsedSqlNode(SCF_SELECT);
424       if ($2 != nullptr) {
425         $$->selection.attributes.swap(*$2);
426         delete $2;
427       }
428       if ($5 != nullptr) {
429         $$->selection.relations.swap(*$5);
430         delete $5;
431       }
432       $$->selection.relations.push_back($4);
433       std::reverse($$->selection.relations.begin(), $$->selection.relations.end());
434
435       if ($6 != nullptr) {
436         $$->selection.conditions.swap(*$6);
437         delete $6;
438       }
439       free($4);
440     }
441     ;
```

图 2. select_stmt 语法解析树

在语法规则中，由空格分割的每一个部分视为一个参数，如图 2 所示，将 SELECT 语句的语法通过空格分割后可以得到 6 个部分。根据 SELECT 语句一般的语法：

SELECT attr FROM rel WHERE ...

理论上来说应该只有 5 个参数，但在该代码中的 FROM 和 WHERE 之间是 ID 和 rel_list 而不只是一个 rel。考虑到 FROM 后面可以接多个表的情况，为了便于后续的操作，在解析阶段需要把多个表进行区分，实际上这里的 rel_list 是一个递归解析结构，在 yacc_sql.y 中查找 `rel_list:` 关键词：

```

546 rel_list:
547     /* empty */
548     {
549         $$ = nullptr;
550     }
551     | COMMA ID rel_list {
552         if ($3 != nullptr) {
553             $$ = $3;
554         } else {
555             $$ = new std::vector<std::string>;
556         }
557
558         $$->push_back($2);
559         free($2);
560     }
561     ;

```

图 3 rel_list 解析结构

在图 3 中，547-550 行是当 rel_list 为空的情况下的解析结果，也就是对应的查询单表的情况：

SELECT select_attr FROM ID WHERE

551-560 行是当 rel_list 不为空的情况下的解析结果，当查询中有多个表时，FROM 后面的表之间通过逗号(,)进行分隔，那么有：

ID rel_list ⇔ ID,ID rel_list' ⇔ ID,ID,ID rel_list'' ⇔ ID,ID,ID,ID rel_list''' ⇔

每一次解析都会将 rel_list 中的第一个 ID 解析出来，然后多次递归直至 rel_list 为空，那么就可以把多个 ID(也就是表名)区分并存储起来，便于后续的处理。以下方 select 语句为例：

SELECT id,name FROM student,course,department

此时解析和递归过程如图 4 所示。

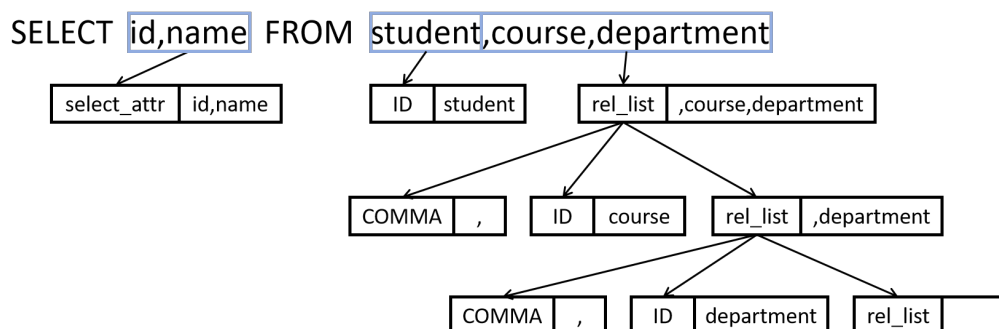
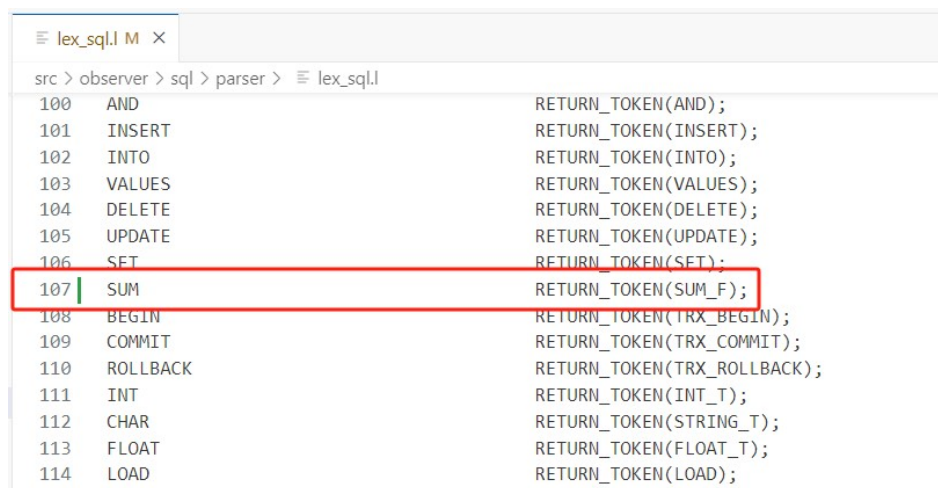


图 4 select 语句递归解析实例

事实上，在图 4 中，select_attr 也是一个递归结构，感兴趣的同学可以在 yacc_sql.y 中查找 select_attr 关键词进行分析，对比一下 select_attr 和 ID rel_list 这两个位置递归的写法。接下来，我们将正式进入 SUM 聚合函数的实现。

0.1 TOKEN 的定义

token 的定义部分在 `./src/observer/sql/parser/lex_sql.l` 中可以找到，例如 `select` 会被解析为 `SELECT` 标识，逗号(,)会被解析为 `COMMA` 标识等等。在语法解析中，token 是不区分大小写的，即输入的 `select` 和 `SELECT` 效果上是等同的，但解析返回的 `SELECT` 标识会区分大小写，规范的写法中标识一般都写成大写。首先在 `lex_sql.l` 中添加 `SUM` 的 token 定义，以将 `SUM` 解析为 `SUM_F` 标识为例，如图 5 所示。



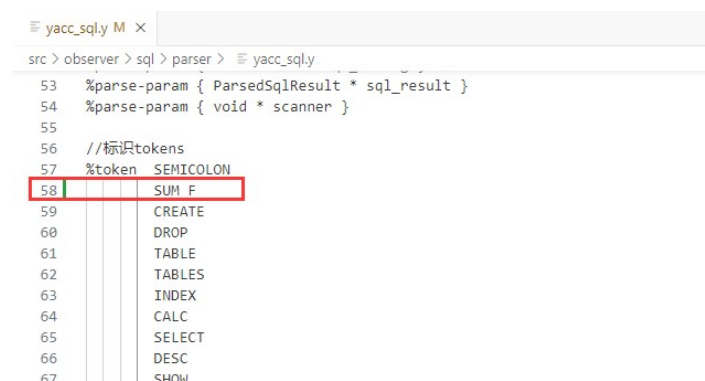
```
src > observer > sql > parser > lex_sql.l
100 AND RETURN_TOKEN(AND);
101 INSERT RETURN_TOKEN(INSERT);
102 INTO RETURN_TOKEN(INTO);
103 VALUES RETURN_TOKEN(VALUE);
104 DELETE RETURN_TOKEN(DELETE);
105 UPDATE RETURN_TOKEN(UPDATE);
106 SET RETURN_TOKEN(SET);
107 SUM RETURN_TOKEN(SUM_F);
108 BEGIN RETURN_TOKEN(TRX_BEGIN);
109 COMMIT RETURN_TOKEN(TRX_COMMIT);
110 ROLLBACK RETURN_TOKEN(TRX_ROLLBACK);
111 INT RETURN_TOKEN(INT_T);
112 CHAR RETURN_TOKEN(STRING_T);
113 FLOAT RETURN_TOKEN(FLOAT_T);
114 LOAD RETURN_TOKEN(LOAD);
```

图 5 SUM 匹配规则

词法分析之后会使用匹配的结果进行下一步操作，在解析步骤中将使用 `SUM_F` 作为 `SUM` 的标识，所以请务必记住此处设置的标识。

0.2 解析规则的定义

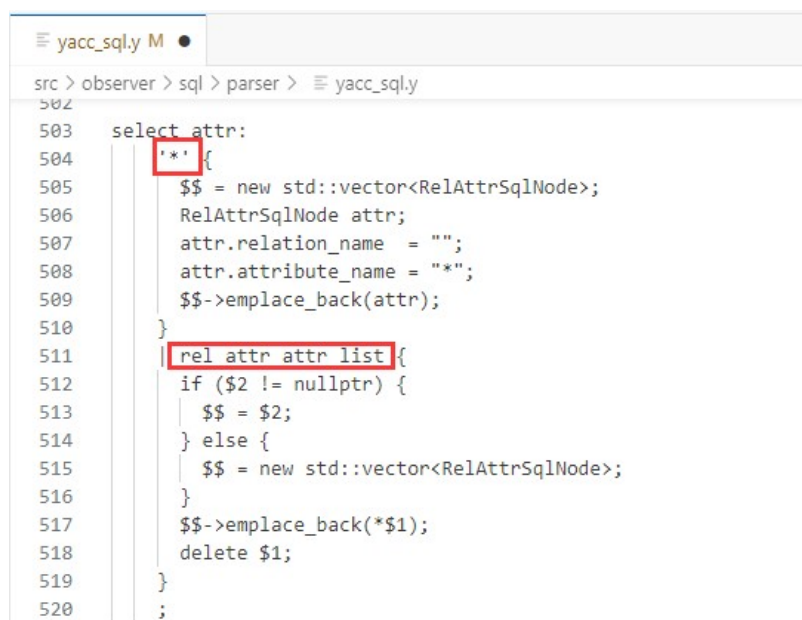
打开 `yacc_sql.y` 文件，在 `lex_sql.l` 中声明匹配规则之后，首先在 `token` 部分添加标识再继续后续步骤，如图 6 所示。



```
src > observer > sql > parser > yacc_sql.y
53 %parse-param { ParsedSqlResult * sql_result }
54 %parse-param { void * scanner }
55
56 //标识tokens
57 %token SEMICOLON
58 SUM_F
59 CREATE
60 DROP
61 TABLE
62 TABLES
63 INDEX
64 CALC
65 SELECT
66 DESC
67 SHOW
```

图 6 SUM_F 标识 token 声明

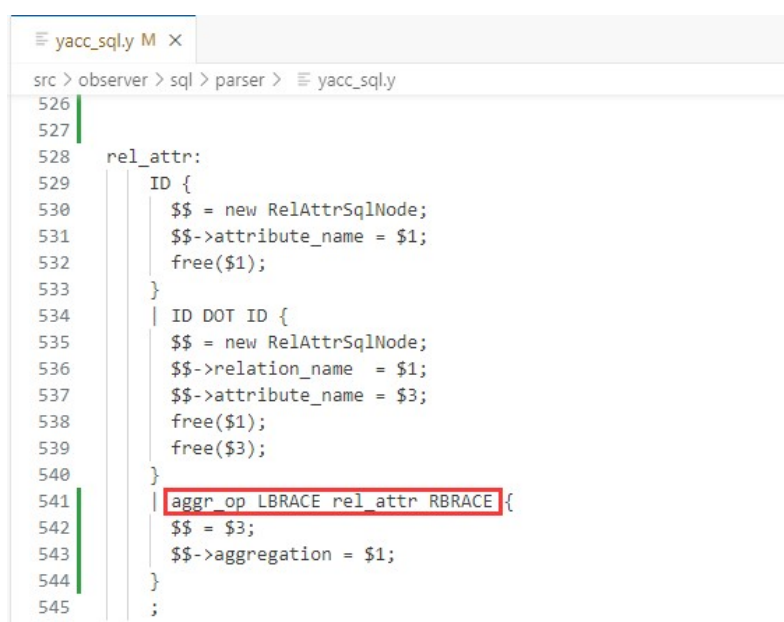
随后，找到 `select_stmt` 解析树部分，我们将在这基础上进行改动来支持 SUM 聚合函数的解析。添加聚合函数的解析，本质上就是从 `select_attr` 开始修改，对其中的属性增加一个聚合的标识，在后续处理时根据聚合标识来进行判断。在 `select_attr` 中，有两个解析分支，如图 7 所示。



```
src > observer > sql > parser > yacc_sql.y
502
503 select_attr:
504     .* {
505         $$ = new std::vector<RelAttrSqlNode>;
506         RelAttrSqlNode attr;
507         attr.relation_name = "";
508         attr.attribute_name = "*";
509         $$->emplace_back(attr);
510     }
511     rel attr attr list {
512         if ($2 != nullptr) {
513             $$ = $2;
514         } else {
515             $$ = new std::vector<RelAttrSqlNode>;
516         }
517         $$->emplace_back(*$1);
518         delete $1;
519     }
520 ;
```

图 7 `select_attr` 解析树

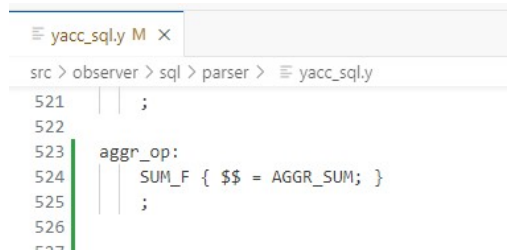
在介绍 FROM 和 WHERE 之间的内容时提到 `select_attr` 也包含一个递归解析，即 `attr_list`。此时的 `rel_attr` 和 `attr_list` 仅能解析单个或多个字段，下一步将通过修改 `rel_attr` 和 `attr_list` 的解析树定义来实现单个或多个字段的聚合功能。找到 `rel_attr` 部分，根据聚合函数的语法，添加如图 8 中所示的语法规则。



```
src > observer > sql > parser > yacc_sql.y
526
527
528 rel_attr:
529     ID {
530         $$ = new RelAttrSqlNode;
531         $$->attribute_name = $1;
532         free($1);
533     }
534     ID DOT ID {
535         $$ = new RelAttrSqlNode;
536         $$->relation_name = $1;
537         $$->attribute_name = $3;
538         free($1);
539         free($3);
540     }
541     aggr_op LBRACE rel_attr RBRACE {
542         $$ = $3;
543         $$->aggregation = $1;
544     }
545 ;
```

图 8 聚合函数语法规则

其中 `aggr_op` 为函数类型的解析(`sum,avg,max...`)，还需要再写一个 `aggr_op` 的解析，在图 5 定义的 `token` 中，我们将字符串“`sum`”匹配为 `SUM_F`，因此 `aggr_op` 部分如图 9 所示。

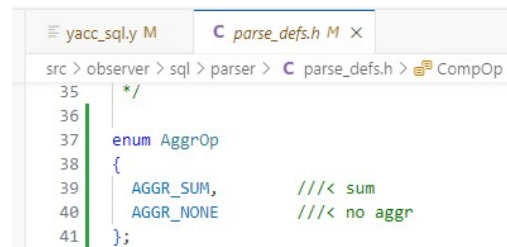


```
521 | ;
522 |
523 | aggr_op:
524 |     SUM_F { $$ = AGGR_SUM; }
525 | ;
526 |
```

图 9 聚合函数类型解析

`AGGR_SUM` 为解析返回给程序后续使用的常量，由于目前还没有关于 `AGGR_SUM` 的定义，接下来将介绍如何对其进行定义。

`./src/observer/sql/parser/parse_defs.h` 文件中声明了不同类型节点的结构，例如选择节点结构，属性节点结构等等，不同节点类的结构中所包含的成员也是不一样的，这些成员一般用来存储语法解析步骤中得到的结果。与此同时，该文件中也声明了一些后续便于在程序中进行判断分析的常量，例如 `CompOp`，其中声明了一些比较运算符，在判断阶段可以根据对应的比较运算符标识来进行对应的比较运算操作。类似地，在该文件中进行聚合函数类型的声明，参照 `CompOp` 的写法，`AggrOp` 的定义如图 10 所示。



```
35 | */
36 |
37 | enum AggrOp
38 | {
39 |     AGGR_SUM,          ///< sum
40 |     AGGR_NONE          ///< no aggr
41 | };
```

图 10 AggrOp

此外，我们在字段结构中还增加了一个表示聚合函数类型的成员，找到 `RelAttrSqlNode` 结构体，在其添加 `aggregation` 成员变量，如图 11 所示。



```
42 |
43 | struct RelAttrSqlNode
44 | {
45 |     std::string relation_name;  ///< relation name (may be NULL) 表名
46 |     std::string attribute_name; ///< attribute name 属性名
47 |     AggrOp aggregation = AGGR_NONE; ///< aggregation (may be empty) 聚合操作
48 | };
```

图 11 RelAttrSqlNode 结构体

在完成了语法解析部分的工作后，对 `yaac_sql.y` 和 `lex_sql.l` 进行编译生成新的 `yaac_sql.cpp`、`yacc_sql.hpp`、`lex_sql.cpp` 和 `lex_sql.h` 文件（这四个文件不要手动进行修改），编译脚本位于 `./src/observer/sql/parser/gen_parser.sh`。在 `miniob` 项目文件中已经配置好了在 `vscode` 中快捷运行部分指令，可以在 `./vscode/tasks.json` 文件中找到，如图 12 所示。

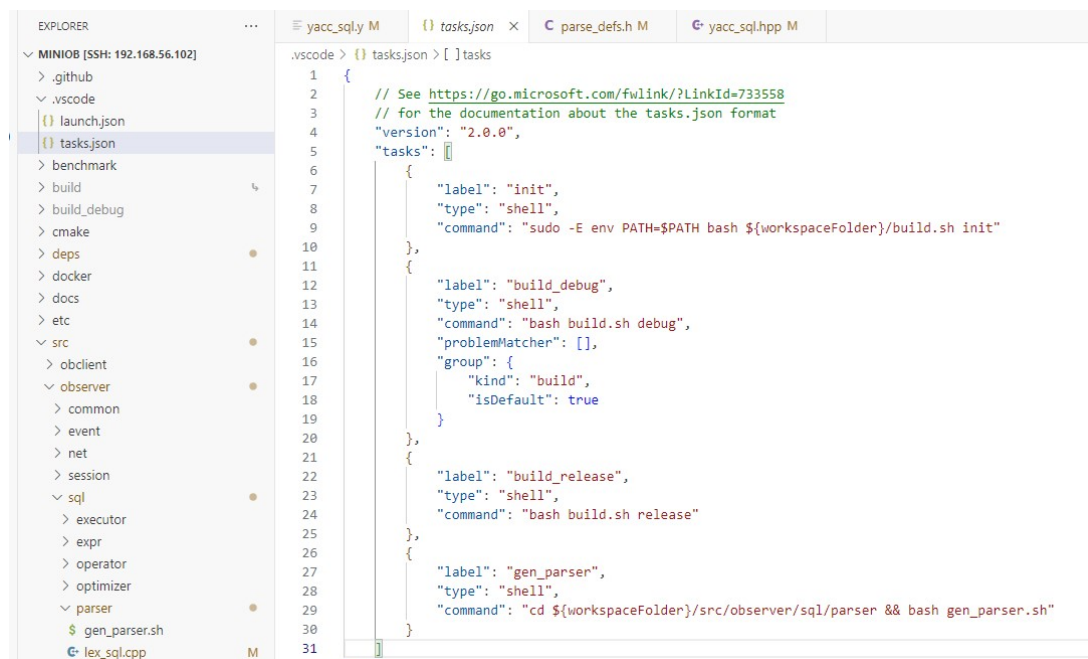


图 12 tasks.json 文件

可以看到 `gen_parser` 已经被写到了快捷运行配置中，因此还可以直接通过 `vscode` 的 `Tasks: Run Task` 运行。（`Ctrl+Shift+P` 弹出搜索框，输入 `Run Task` 后选择搜索到的 `Tasks: Run Task`，点进去后即可看到 `tasks.json` 中配置的任务标签，选择 `gen_parser`）

编译完语法解析部分后，可以对整个项目重新编译来测试聚合函数的解析功能。与编译语法解析一样，在 `Tasks: Run Task` 列表中选择 `build_debug` 即可对整个项目进行编译。

直接编译可能会有编译过程卡死甚至整个 Linux 系统直接崩溃重启的现象，因为官方的脚本默认并发编译，且会自动判断并发数，但实际情况下 CPU 和内存并不匹配。解决方案是在编译时调整或去掉并发，在命令后加上 `--make -jN` 参数，`N` 自行进行调整(`N` 为并发数)，例如：

```
sudo bash build.sh debug --make -j1
```

当然，如果主机配置比较好，给虚拟机分配大于 8G 的内存（越大越好）可以不用改并发数，直接解决问题。

编译完成后，进入 build_debug 目录下，输入：

```
./bin/observer -f ../etc/observer.ini -P cli
```

即可启动程序，更多运行细节可以查阅[官方文档](#)。在语法解析阶段，我们已经把语句解析成了程序可以理解的形式，接下来是程序对解析的结果进行处理的过程。

1. Resolve 阶段

在 Parse 阶段结束后，解析程序将返回一个 `ParsedSqlNode` 类的变量，该变量中存储着解析得到的内容，在 `parse_defs.h` 中可以查看 `ParsedSqlNode` 的定义。Parse 阶段的下一阶段为 `Resolve`，从字面意思来说，`Resolve` 的含义为“解析，解决”，表示处理得到结果的过程，如果说 `Parse` 阶段是对 `sql` 语句的语法进行处理，那么 `Resolve` 可以近似地看作是 `Parse` 结果的后处理。在 `ParsedSqlNode` 的定义中，我们可以看出它包含了所有可能的节点结构(`select`, `insert`, `update`, etc)，如果将其直接传递给后续的处理过程中显然不够优雅，`Resolve` 阶段就是对 `ParsedSqlNode` 进一步处理从而得到更具体的节点类型的过程，在数据库中，解析得到的最终结果被称为“`Statement`”，缩写“`Stmt`”。从 `session_stage.cpp` 中找到 `resolve_stage` 的部分，可以找到创建 `Stmt` 的入口，如图 13 所示。

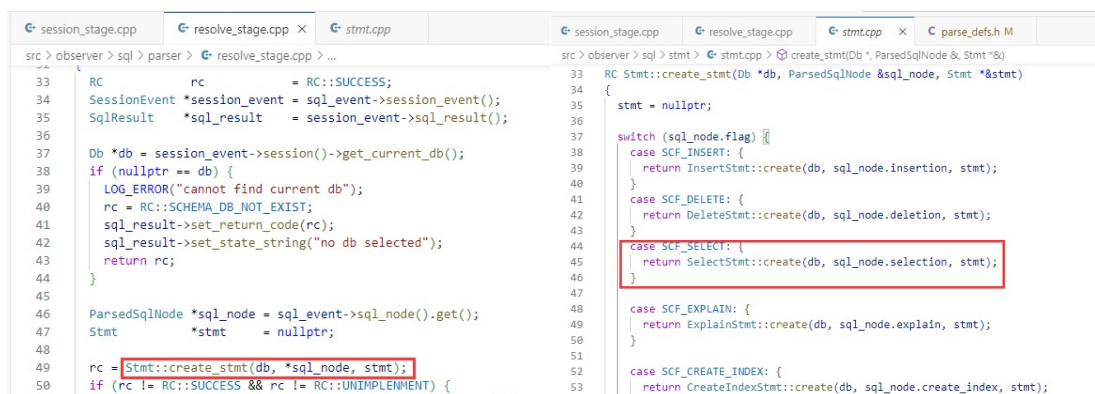
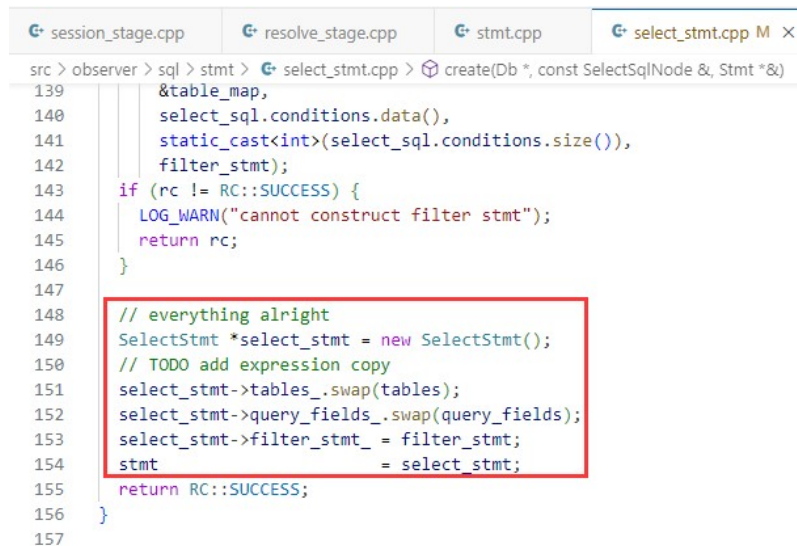


图 13 Stmt 的创建

不同操作类型(`select`, `insert`, etc.)对应着不同的类，但所有类都是由 `Stmt` 类继承，并对 `create_stmt` 函数进行重载而来的。进入 `SelectStmt::create` 函数部分，看到最后函数返回的地方，如图 14 所示。



```

src > observer > sql > stmt > select_stmt.cpp > create(Db *, const SelectSqlNode &, Stmt *&)
139     &table_map,
140     select_sql.conditions.data(),
141     static_cast<int>(select_sql.conditions.size()),
142     filter_stmt);
143     if (rc != RC::SUCCESS) {
144         LOG_WARN("cannot construct filter stmt");
145         return rc;
146     }
147
148     // everything alright
149     SelectStmt *select_stmt = new SelectStmt();
150     // TODO add expression copy
151     select_stmt->tables_.swap(tables);
152     select_stmt->query_fields_.swap(query_fields);
153     select_stmt->filter_stmt_ = filter_stmt;
154     stmt = select_stmt;
155     return RC::SUCCESS;
156 }
157

```

图 14 select_stmt 的创建

从代码上来看，select_stmt 需要传递的参数有 tables, query_fields 和 filter_stmt_，而前部分的代码就是在从 parse 得到的内容中得到这些参数。前面提到聚合函数是 field 字段的一部分，为了添加聚合函数的支持，就需要从 query_fields 逆向分析，在每一个 field 中添加聚合函数的标识。找到 query_fields 的处理代码，可以找到每一个 field 的定义入口，如图 15 所示。



```

src > observer > sql > stmt > select_stmt.cpp > create(Db *, const SelectSqlNode &, Stmt *&)
110     if (tables.size() != 1) {
111         LOG_WARN("invalid. I do not know the attr's table. attr=%s", relation_attr.attribute_name.c_str());
112         return RC::SCHEMA_FIELD_MISSING;
113     }
114
115     Table *table = tables[0];
116     const FieldMeta *field_meta = table->table_meta().field(relation_attr.attribute_name.c_str());
117     if (nullptr == field_meta) {
118         LOG_WARN("no such field. field=%s.%s.%s", db->name(), table->name(), relation_attr.attribute_name.c_str());
119         return RC::SCHEMA_FIELD_MISSING;
120     }
121
122     query_fields.push_back(Field(table, field_meta));
123 }
124
125

```

图 15 query_fields 的定义

进入 Field 类中，参照类成员 table_ 和 field_ 的写法，添加聚合函数标识的变量成员及必要的成员函数，最终 Field 类的修改结果如图 16 所示。

```

25 class Field
26 {
27 public:
28     Field() = default;
29     // Field(const Table *table, const FieldMeta *field) : table_(table), field_(field) {}
30     Field(const Table *table, const FieldMeta *field, const AggrOp aggregation=AggrOp::AGGR_NONE) : table_(table), field_(field), aggregation_(aggregation) {}
31     Field(const Field &) = default;
32
33     const Table *table() const { return table_; }
34     const FieldMeta *meta() const { return field_; }
35
36     AttrType attr_type() const { return field_>type(); }
37
38     const char *table_name() const { return table_>name(); }
39     const char *field_name() const { return field_>name(); }
40     const AggrOp aggregation() const { return aggregation_; }
41
42     void set_table(const Table *table) { this->table_ = table; }
43     void set_field(const FieldMeta *field) { this->field_ = field; }
44
45     void set_int(Record &record, int value);
46     int get_int(const Record &record);
47
48     const char *get_data(const Record &record);
49
50 private:
51     const Table *table_ = nullptr;
52     const FieldMeta *field_ = nullptr;
53     AggrOp aggregation_ = AggrOp::AGGR_NONE;
54 };

```

图 16 Field 类

值得注意的是，在解析步骤字段不一定会使用聚合函数，所以需要设定默认值为无聚合操作，便于后续处理进行判断。对 Field 类修改完成后，再回到 select_stmt 的定义部分增加聚合函数参数的传递，如图 17 所示。

```

110 if (tables.size() != 1) {
111     LOG_WARN("invalid. I do not know the attr's table. attr=%s", relation_attr.attril
112     return RC::SCHEMA_FIELD_MISSING;
113 }
114
115 Table *table = tables[0];
116 const FieldMeta *field_meta = table->table_meta().field(relation_attr.attribute_nam
117 if (nullptr == field_meta) {
118     LOG_WARN("no such field. field=%s.%s.%s", db->name(), table->name(), relation_att
119     return RC::SCHEMA_FIELD_MISSING;
120 }
121
122 const AggrOp aggregation_ = relation_attr.aggregation;
123
124 query_fields.push_back(Field(table, field_meta, aggregation_));
125 }
126 }

```

图 17 query_fields 部分的修改

到目前为止，关于 stmt 的定义已经完成，接下来将在 stmt 的基础上进行算子的定义，而算子是数据库系统执行的依据。

2. Optimize 阶段

在 SQL 语句的执行过程中，可以分为多个不同的执行步骤，而每一个步骤就会包含一个或者多个算子。算子可以看作是一个执行函数，例如以下 SQL 语句：

SELECT select_attr FROM ID WHERE

该语句中至少包含 TableScan, Filter, Projection 和 From 算子(From 一般隐式地包含在了 TableScan 中, 不会特意地声明), 如果涉及到索引, 还会包含 IndexScan。考虑最简单的情况, select 语句中算子的执行顺序如图 18 所示。



图 18 select 语句中算子的执行顺序

聚合同样作为一个算子, 首先需要定义聚合算子的具体执行过程, 然后将该算子添加到 select 语句中算子执行序列的合适位置即可完成聚合操作。进入 optimize_stage.cpp 的代码部分, 按照和 Resolve 阶段一样的逻辑进行分析, 找到创建算子的入口, 如图 19 所示。

```
src > observer > sql > optimizer > optimize_stage.cpp > ...
32 RC OptimizeStage::handle_request(SQLStageEvent *sql_event)
33 {
34     unique_ptr<LogicalOperator> logical_operator;
35
36     RC rc = create_logical_plan(sql_event, logical_operator);
37     if (rc != RC::SUCCESS) {
38         if (rc != RC::UNIMPLEMENTED) {
39             LOG_WARN("failed to create logical plan. rc=%s", strrc(rc));
40         }
41         return rc;
42     }
43
44     rc = rewrite(logical_operator);    重写
45     if (rc != RC::SUCCESS) {
46         LOG_WARN("failed to rewrite plan. rc=%s", strrc(rc));
47         return rc;
48     }
49
50     rc = optimize(logical_operator);  优化
51     if (rc != RC::SUCCESS) {
52         LOG_WARN("failed to optimize plan. rc=%s", strrc(rc));
53         return rc;
54     }
55
56     unique_ptr<PhysicalOperator> physical_operator;
57     rc = generate_physical_plan(logical_operator, physical_operator);
58     if (rc != RC::SUCCESS) {
59         LOG_WARN("failed to generate physical plan. rc=%s", strrc(rc));
60         return rc;
61     }
62
63     sql_event->set_operator(std::move(physical_operator));
64
65     return rc;
66 }
```

图 19 optimize 阶段

在 optimize 阶段中, 可以发现 logical_operator 和 physical_operator 操作, 而在逻辑操作和物理操作之间有两个以逻辑操作为参数的函数, 即 rewrite 和 optimize 函数(这也是为什么这一阶段称为 optimize 的原因)。逻辑操作是预先定义的操作

步骤，逻辑操作不会被真正地执行，而是经过 **rewrite** 和 **optimize** 之后生成物理操作，物理操作才是数据库系统最终执行的依据。之所以有逻辑和物理两者之分，原因之一是逻辑操作中有可能有可以优化的空间，例如表的连接操作(join)，在有条件语句的情况下，先执行条件语句再连接和先连接再执行条件语句两种情况下性能显然是存在差异的，而 **optimize** 阶段事实上主要目的是为了调节算子实现更高的执行效率，定义算子只是过程而非目的。

首先找到 **select** 语句的逻辑操作定义(optimizer/logical_plan_generator.cpp 文件)，在定义 **select** 逻辑操作的函数中，可以发现 **table_get_oper**, **predicate_oper** 和 **project_oper** 三个逻辑算子，并且逻辑算子类含有一个名为 **add_child** 的成员函数用于连接和执行多个逻辑算子。从 **add_child** 的调用顺序来看，可以得到这三个逻辑算子的执行顺序为：

table_get_oper -> predicate_oper -> project_oper

add_child 可以理解为添加子操作，一般来说子操作完成后才能执行主操作，所以越底层的 **child** 被执行的优先级越高。

聚合是对最终得到的元组中的某些字段进行的操作，可以很容易地确定聚合算子的位置应该在 **project_oper** 之后，定义聚合算子为 **aggregate_oper**，添加了聚合算子后的算子执行顺序为：

table_get_oper -> predicate_oper -> project_oper -> aggregate_oper

关于 **aggregate_oper** 的定义可以参照其他算子来进行编写，以 **insert** 逻辑算子为例，找到 **insert_logical_operator.h** 和 **insert_logical_operator.cpp**，如图 20 所示。

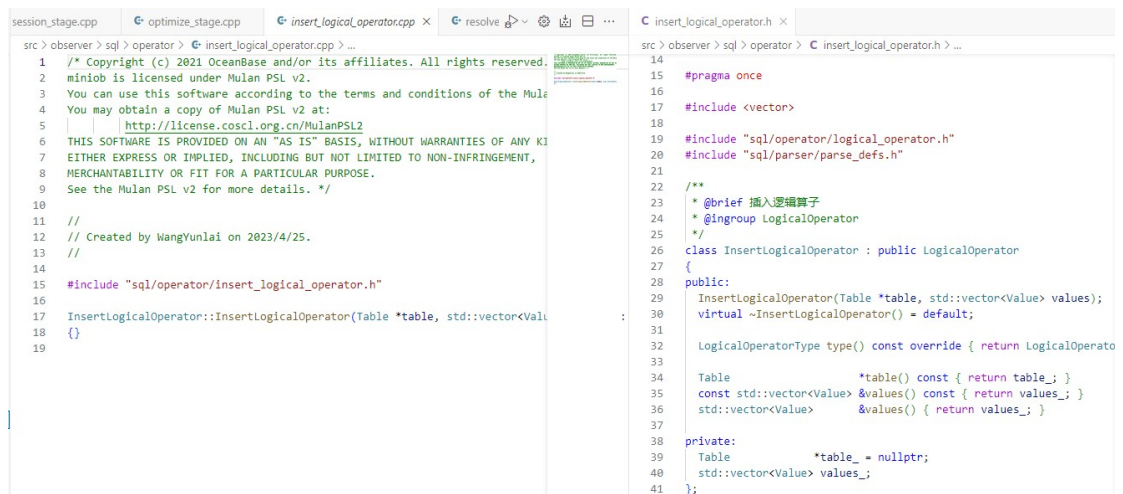


图 20 insert_logical_operator.h/cpp

从成员变量来看，insert 逻辑算子含有 table 和 values 两个成员变量，而 insert 操作中正好涉及到 table 和 value，因此 insert 才这么定义。同理，aggregate 操作涉及的对象为字段和聚合函数类型，但前面我们已经在字段中添加了聚合类型，因此只需要字段即可。参照 insert 逻辑算子的定义，首先在 operator 文件夹下创建 aggregate_logical_operator.cpp 和 aggregate_logical_operator.h 两个文件，逻辑算子的定义如图 21 所示。

```
C aggregate_logical_operator.h U x
src > observer > sql > operator > C aggregate_logical_operator.h > ...
1 #pragma once
2
3 #include <memory>
4 #include <vector>
5
6 #include "sql/expr/expression.h"
7 #include "sql/operator/logical_operator.h"
8 #include "storage/field/field.h"
9
10 class AggregateLogicalOperator: public LogicalOperator{
11 public:
12     AggregateLogicalOperator(const std::vector<Field> &field);
13     virtual ~AggregateLogicalOperator() = default;
14
15     LogicalOperatorType type() const override{
16         return LogicalOperatorType::AGGREGATE;
17     }
18
19     const std::vector<Field> &fields() const{
20         return fields_;
21     }
22
23 private:
24     std::vector<Field> fields_;
25
26 };

C aggregate_logical_operator.cpp U x
src > observer > sql > operator > C aggregate_logical_operator.cpp > ...
1 #include "sql/operator/aggregate_logical_operator.h"
2
3 AggregateLogicalOperator::AggregateLogicalOperator(const std::vector<Field> &field): fields_(field){};
```

图 21 aggregate_logical_operator.h/cpp

在定义完 aggregate 逻辑算子后，将其添加到 select 操作中。回到 select 逻辑操作的定义函数部分，在 project_oper 之后添加 aggregate_oper，如图 22 所示。

```
C session_stage.cpp C optimize_stage.cpp C logical_plan_generator.cpp M x C resolve_stage.cpp C st
src > observer > sql > optimizer > C logical_plan_generator.cpp > create_plan(FilterStmt *, unique_ptr<LogicalOperator> &)
120 if (predicate_oper) {
121     if (table_oper) {
122         predicate_oper->add_child(std::move(table_oper));
123     }
124     project_oper->add_child(std::move(predicate_oper));
125 } else {
126     if (table_oper) {
127         project_oper->add_child(std::move(table_oper));
128     }
129 }
130
131 bool aggr_flag = false;
132 for(auto field:all_fields){
133     if(field.aggregation() != AggrOp::AGGR_NONE){
134         aggr_flag = true;
135         break;
136     }
137 }
138
139 if(aggr_flag){
140     unique_ptr<LogicalOperator> aggregate_oper(new AggregateLogicalOperator(all_fields));
141     aggregate_oper->add_child(std::move(project_oper));
142     logical_operator.swap(aggregate_oper);
143 }else{
144     logical_operator.swap(project_oper);
145 }
146
147 // logical_operator.swap(project_oper);
148 return RC::SUCCESS;
149 }
```

图 22 在 select 操作中添加聚合算子

此时 select 操作已经添加了聚合算子，接下来 rewrite 和 optimize 暂时不需要考虑，直接进入物理计划的创建步骤。

与逻辑计划的创建类似，我们找到其他逻辑算子的物理计划创建函数，仿照其他的创建函数写法编写聚合逻辑算子的物理计划创建函数，如图 23 所示。

```
301 RC PhysicalPlanGenerator::create_plan(AggregateLogicalOperator &aggregate_oper, unique_ptr<PhysicalOperator> &oper)
302 {
303     vector<unique_ptr<LogicalOperator>> &children_oper = aggregate_oper.children();
304     ASSERT(children_oper.size() == 1, "aggregate logical operator's sub oper number should be 1");
305
306     LogicalOperator &child_oper = *children_oper.front();
307
308     unique_ptr<PhysicalOperator> child_phy_oper;
309     RC rc = create(child_oper, child_phy_oper);
310     if (rc != RC::SUCCESS) {
311         LOG_WARN("failed to create child operator of predicate operator. rc=%s", strrc(rc));
312         return rc;
313     }
314
315     AggregatePhysicalOperator *aggregate_operator = new AggregatePhysicalOperator;
316     const vector<Field> &aggregate_fields = aggregate_oper.fields();
317     LOG_TRACE("got %d aggregation fields", aggregate_fields.size());
318     for (const Field &field : aggregate_fields) {
319         aggregate_operator->add_aggregation(field.aggregation());
320     }
321
322     if (child_phy_oper) {
323         aggregate_operator->add_child(std::move(child_phy_oper));
324     }
325
326     oper = unique_ptr<PhysicalOperator>(aggregate_operator);
327
328     LOG_TRACE("create an aggregate physical operator");
329     return rc;
330 }
```

图 23 聚合物理计划的创建

接下来进入聚合物理算子的定义步骤。物理算子的定义和逻辑算子的定义类似，接下来将以 delete 物理算子为例作为参照(不选择 insert 的原因在于 insert 没有涉及到子操作，而 delete 操作有可能会涉及到 filter 操作)。打开 delete_physical_operator.cpp 和 delete_physical_operator.h，如图 24 所示。

```
src > observer > sql > operator > delete_physical_operator.cpp > ...
30     return rc;
31 }
32
33 trx_ = trx;
34
35 return RC::SUCCESS;
36 }
37
38 RC DeletePhysicalOperator::next()
39 {
40     RC rc = RC::SUCCESS;
41     if (children.empty()) {
42         return RC::RECORD_EOF;
43     }
44
45     PhysicalOperator *child = children[0].get();
46     while (RC::SUCCESS == (rc = child->next())) {
47         Tuple *tuple = child->current_tuple();
48         if (nullptr == tuple) {
49             LOG_WARN("failed to get current record: %s", strrc(rc));
50             return rc;
51         }
52
53         RowTuple *row_tuple = static_cast<RowTuple*>(tuple);
54         Record &record = row_tuple->record();
55         rc = trx_>delete_record(table_, record);
56         if (rc != RC::SUCCESS) {
57             LOG_WARN("failed to delete record: %s", strrc(rc));
58             return rc;
59         }
60     }
61
62     return RC::RECORD_EOF;
63 }
64
```

```
src > observer > sql > operator > delete_physical_operator.h > ...
14
15 #pragma once
16
17 #include "sql/operator/physical_operator.h"
18
19 class Trx;
20 class DeleteStat;
21
22 /**
23  * @brief 物理算子，删除
24  * @ingroup PhysicalOperator
25  */
26 class DeletePhysicalOperator : public PhysicalOperator
27 {
28 public:
29     DeletePhysicalOperator(Table *table) : table_(table) {}
30
31     virtual ~DeletePhysicalOperator() = default;
32
33     PhysicalOperatorType type() const override { return PhysicalOperatorType::DELETE; }
34
35     RC open(Trx *trx) override;
36     RC next() override;
37     RC close() override;
38
39     Tuple *current_tuple() override { return nullptr; }
40
41 private:
42     Table *table_ = nullptr;
43     Trx *trx_ = nullptr;
44 };
45
```


图 24 delete_physical_operator.cpp/h

在物理算子类中，主要有 `open()`, `next()` 和 `close()` 三个成员函数，其中 `open()` 为算子的初始化，`next()` 为初始化之后剩余需要执行的内容，`close()` 为算子执行结束后需要额外执行的内容。在 `delete_physical_operator.cpp` 中，`open()` 实际上没有进行操作，只是为成员变量进行了赋值；`next()` 为实际执行了 `delete` 操作的部分，根据 `delete` 语句的执行逻辑，可以推测 `delete` 算子之前大概率是 `filter` 算子，但无论是什么算子，都会返回一组元组，而多个元组是一条一条地返回的，每执行一次 `next()` 就会跳转到下一条返回的元组。在 `delete` 中，首先获取了子节点(子操作)，然后调用子节点的 `next()` 来依次获得元组，由于这些元组都是符合条件的需要被 `delete` 的元组，因此 `while` 语句中直接删除元组对应的记录即可。为了更好地理解物理算子如何进行定义，可以再分析一下其他物理算子的定义代码(`insert`, `select`, etc.)。

接下来，我们将定义 `aggregate` 操作的物理算子。`aggregate` 算子不可单独执行，需要在 `project` 算子之后才能执行，因此 `aggregate` 算子拥有子节点，这一点上与 `delete` 算子十分相似。对于 `sum` 函数，其结果为所有元组中对应字段值的和，可以想到每遍历一个元组，将其元组中对应字段的值加到一个累加变量中，当遍历结束时累加变量的值即为最终的结果。在 `operator` 文件夹下新建 `aggregate_physical_operator.cpp` 和 `aggregate_physical_operator.h` 文件，`aggregate` 操作的物理算子定义如图 25 所示。

```

src > observer > sql > operator > C aggregate_physical_operator.h > ...
1 #pragma once
2
3 #include "sql/operator/physical_operator.h"
4 #include "sql/parser/parse.h"
5 #include "sql/expr/tuple.h"
6
7 /**
8  * @brief 聚合物理算子
9  * @ingroup PhysicalOperator
10 */
11 class AggregatePhysicalOperator : public PhysicalOperator
12 {
13 public:
14     AggregatePhysicalOperator(){}
15
16     virtual ~AggregatePhysicalOperator() = default;
17
18     void add_aggregation(const AggrOp aggregation);
19
20     PhysicalOperatorType type() const override
21     {
22         return PhysicalOperatorType::AGGREGATE;
23     }
24
25     RC open(Trx *trx) override;
26     RC next() override;
27     RC close() override;
28
29     Tuple *current_tuple() override;
30
31 private:
32     std::vector<AggrOp> aggregations_;
33     ValueListTuple result_tuple_;
34 };

```

```

src > observer > sql > operator > C aggregate_physical_operator.cpp > next()
19 RC AggregatePhysicalOperator::next()
20 {
21     // already aggregated
22     if (result_tuple_.cell_num() > 0) {
23         return RC::RECORD_EOF;
24     }
25
26     RC rc = RC::SUCCESS;
27     PhysicalOperator *oper = children_[0].get();
28
29     std::vector<Value> result_cells;
30     while (RC::SUCCESS == (rc = oper->next())) {
31         // get tuple
32         Tuple *tuple = oper->current_tuple();
33
34         // do aggregate
35         for (int cell_idx = 0; cell_idx < (int)aggregations_.size(); cell_idx++) {
36             const AggrOp aggregation = aggregations_[cell_idx];
37
38             Value cell;
39             AttrType attr_type = AttrType::INTS;
40             switch (aggregation) {
41                 case AggrOp::AGGR_SUM:
42                     rc = tuple->cell_at(cell_idx, cell);
43                     attr_type = cell.attr_type();
44                     if (attr_type == AttrType::INTS or attr_type == AttrType::FLOATS) {
45                         result_cells[cell_idx].set_float(result_cells[cell_idx].get_float() + cell.get_float());
46                     }
47                     break;
48                 default:
49                     return RC::UNIMPLEMENT;
50             }
51         }
52
53         if (rc == RC::RECORD_EOF) {
54             rc = RC::SUCCESS;
55         }
56
57         result_tuple_.set_cells(result_cells);
58
59         return rc;
60     }
61 }

```

图 25 aggregate 物理算子的定义

这里主要展示 `next()` 函数的定义，`open()` 和 `close()` 没有什么需要特意修改的地方，可以直接参照其他算子的写法。值得注意的是，在类的成员变量中，我们增加了一个名为 `result_tuple_` 的变量，在 `next()` 中用作 `aggregate` 操作是否完成的标记。

3. Execute 阶段

执行阶段将依据 `optimize` 阶段得到的物理算子进行执行和处理，该部分暂时没有需要修改的地方，因此到目前为止，聚合函数(`sum` 函数)的功能已经基本完成。对整个项目工程进行重新编译测试，可得初步的本地测试结果如图 26 所示。

```
miniob > select * from test2;
id | value
1 | 1.4
2 | 4.5

miniob > select sum(id) from test2;
id
3

miniob > select sum(id),sum(value) from test2;
id | value
3 | 5.9

miniob > select sum(value) from test2;
value
5.9
```

图 26 sum 聚合函数测试

4. 本地测试补充

完成了 `sum`, `avg`, `max`, `min` 和 `count` 的聚合函数功能之后，在本地测试中发现还需要处理许多聚合相关的细节，这里列举出了除了实现基本的聚合功能之外，还需要额外修改的内容：

- (1) `*` 只能和 `count` 函数配合使用，即 `count(*)` 是合法的，而其他的例如 `min(*)`, `avg(*)` 都是非法的；
- (2) 聚合函数中的属性只能有一个属性名，`avg(id)` 是合法的，而 `avg(id, value)` 是非法的；
- (3) 聚合函数中的属性不能是空，即 `avg()`, `count()` 是非法的；
- (4) 使用聚合函数后输出的表格列名需要更改，即：

```
miniob > select * from test2;
id | value
1 | 1.4
2 | 4.5

miniob > select sum(id) from test2;
id
3

miniob > select sum(id),sum(value) from test2;
id | value
3 | 5.9

miniob > select sum(value) from test2;
value
5.9

miniob > select * from test;
id | value
0 | 12.1
1 | 14.2

miniob > select sum(*) from test;
FAILURE

miniob > select sum(id), avg(value) from test;
SUM(id) | AVG(value)
1 | 13.15
```

(左侧输出的是原表中的属性名，而聚合之后输出的列名是 `FUNC(属性名)`)

4.1 处理聚合函数中多属性以及空值的非法样例(针对问题(2),(3))

在语法解析阶段，我们只考虑到了聚合函数中包含单个属性字段的情况，而测试程序中支持聚合函数包含多个属性和空字段的解析，而在后续的处理中再返回 FAILURE。首先找到语法解析部分，和 select 多属性名的解析思路一样，我们在聚合函数的括号中匹配属性的地方增加多属性和空值的情况，如图 27 所示。

```
541 rel_attr:
542 ID {
543     $$ = new RelAttrSqlNode;
544     $$->attribute_name = $1;
545     free($1);
546 }
547 ID DOT ID {
548     $$ = new RelAttrSqlNode;
549     $$->relation_name = $1;
550     $$->attribute_name = $3;
551     free($1);
552     free($3);
553 }
554 aggr_op LBRACE rel_attr_aggr rel_attr_aggr_list RBRACE {
555     $$ = $3;
556     $$->aggregation = $1;
557     // redundant columns
558     if ($4 != nullptr){
559         $$->valid = false;
560         delete $4;
561     }
562 }
563 aggr_op LBRACE RBRACE {
564     $$ = new RelAttrSqlNode;
565     $$->relation_name = "";
566     $$->attribute_name = "";
567     $$->aggregation = $1;
568     // empty columns
569     $$->valid = false;
570 }
571 ;
```

图 27 解析多属性字段以及空值情况

当 rel_attr_aggr_list 非空时，表示聚合函数中输入了多个属性名，则此时非法。考虑在解析阶段直接对这两种非法的输入进行标记，在 RelAttrSqlNode 结构体中添加 valid 字段，表示其聚合操作是否合法，如图 28 所示。

```
48 struct RelAttrSqlNode
49 {
50     std::string relation_name;    ///< relation name (may be NULL) 表名
51     std::string attribute_name;   ///< attribute name 属性名
52     AggrOp aggregation = AGGR_NONE; ///< aggregation (may be empty) 聚合操作
53     bool valid = true;           ///< 判断聚合是否合法
54 };
```

图 28 加入合法标识

此外，在 中，rel_attr_aggr 是一个新的语法解析树，匹配普通字段以及*，定义如图 29 所示。

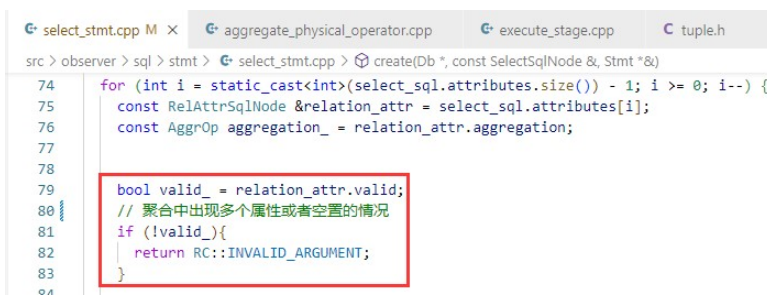
```

rel_attr_aggr:
  '*' {
    $$ = new RelAttrSqlNode;
    $$->relation_name = "";
    $$->attribute_name = "*";
  }
  ID {
    $$ = new RelAttrSqlNode;
    $$->attribute_name = $1;
    free($1);
  } | ID DOT ID {
    $$ = new RelAttrSqlNode;
    $$->relation_name = $1;
    $$->attribute_name = $3;
    free($1);
    free($3);
  }

```

图 29 rel_attr_aggr 语法解析树

有了 valid 标记之后，我们可以在构建 select_stmt 时进行判断，当在语法解析阶段出现了这两种非法操作时，程序直接返回错误。在 select_stmt 的创建函数中的修改如图 30 所示。



```

74 for (int i = static_cast<int>(select_sql.attributes.size()) - 1; i >= 0; i--) {
75     const RelAttrSqlNode &relation_attr = select_sql.attributes[i];
76     const AggrOp aggregation_ = relation_attr.aggregation;
77
78     bool valid_ = relation_attr.valid;
79     // 聚合中出现多个属性或者空置的情况
80     if (!valid_) {
81         return RC::INVALID_ARGUMENT;
82     }
83 }

```

图 30 根据 valid 初步判断聚合是否合法

4.2 sum(*), avg(*)等非法处理(针对问题(1))

对于该问题的处理，可以在 select_stmt 的创建函数中进行判断。由于我们可以获得每一个 select_attr 的内容，可以判断当属性名为"*"而聚合函数不是 count 时，则程序直接返回错误。一种可行的修改方法如图 31 所示。（注意，在图所示代码的下面还有一个属性名也为*的情况，也需要进行聚合函数的判断，可以搜索 "0 == strcmp(field_name, "*")" ）



```

92 }
93
94 if (common::is_blank(relation_attr.relation_name.c_str()) &&
95     0 == strcmp(relation_attr.attribute_name.c_str(), "*")) { // filed is *
96
97     if (have_aggregation_ && aggregation_ != AggrOp::AGGR_COUNT) {
98         return RC::INVALID_ARGUMENT;
99     }
100
101     for (Table *table : tables) {
102         wildcard_fields(table, query_fields, aggregation_);
103     }

```

图 31 处理非 count(*)的非法聚合

4.3 输出列名修改

信息的输出是在 `execute_stage` 执行的，在打开 `execute_stage.cpp` 文件，可以看到 `StmtType::SELECT` 的 case，如图 32 所示：

```

src > observer > sql > executor > execute_stage.cpp > handle_request_with_physical_operator(SQLStageEvent *)
60
61     unique_ptr<PhysicalOperator> &physical_operator = sql_event->physical_operator();
62     ASSERT(physical_operator != nullptr, "physical operator should not be null");
63
64     // TODO 这里也可以优化一下, 是否可以让physical operator自己设置tuple schema
65     TupleSchema schema;
66     switch (stmt->type()) {
67     case StmtType::SELECT: {
68         SelectStmt *select_stmt = static_cast<SelectStmt *>(stmt);
69         bool with_table_name = select_stmt->tables().size() > 1;
70
71         for (const Field &field : select_stmt->query_fields()) {
72             const AggrOp aggr = field.aggregation();
73             if (with_table_name) {
74                 schema.append_cell(field.table_name(), field.field_name(), aggr);
75             } else {
76                 schema.append_cell(field.field_name(), aggr);
77             }
78         }
79     } break;
80

```

图 32 execute_stage 中 select 的执行内容

对于输出的字段信息，其内容存放在一个名为“schema”的 TupleSchema 类对象中。打开 schema 的 append_cell 方法(的 line 74)，可以看出其加入的是 TupleCellSpec 类对象，如图 33 所示。

```

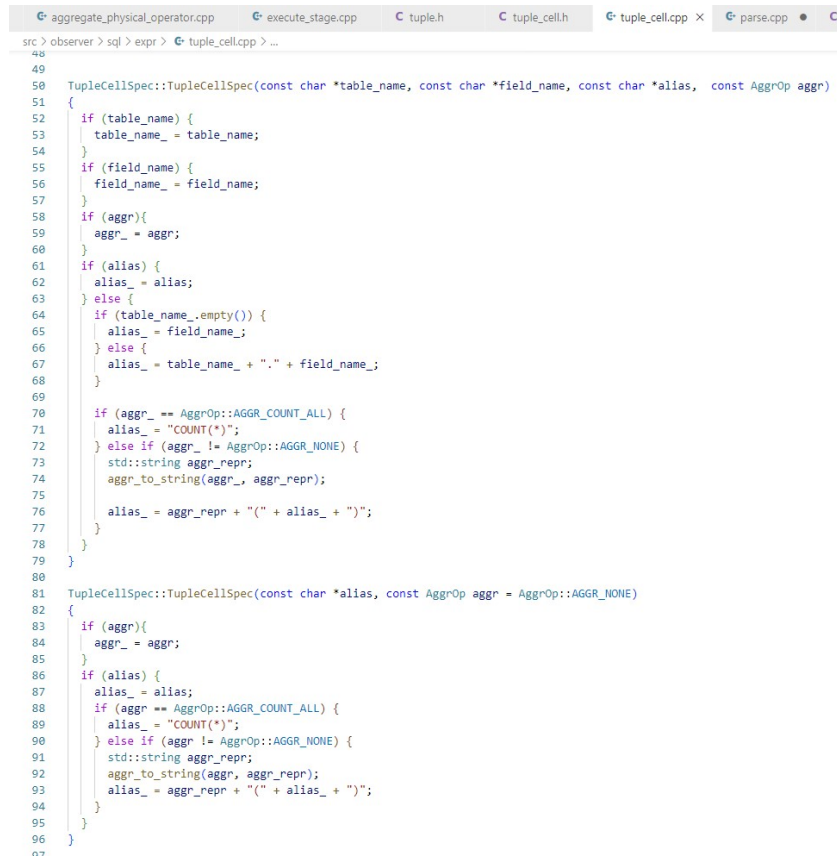
c> aggregate_physical_operator.cpp      execute_stage.cpp      C tuple.h      C tuple_cell.h      C tuple_cell.cpp
c> observer > sql > expr > C tuple.h > TupleSchema > cells_
37  @code { .cpp }
38  * Project(t1.a+t2.b)
39  *
40  * |
41  * Joined
42  * / \
43  * Row(t1) Row(t2)
44  * @endcode
45  */
46
47  /**
48  * @brief 元组的结构，包含哪些字段(这里成为Cell)，每个字段的说明
49  * @ingroup Tuple
50  */
51  class TupleSchema
52  {
53  public:
54      void append_cell(const TupleCellSpec &cell) { cells_.push_back(cell); }
55      void append_cell(const char *table, const char *field, const AggrOp aggr = AGGR_NONE){
56          append_cell(TupleCellSpec(table, field, nullptr, aggr));
57      }
58      void append_cell(const char *table, const char *field) { append_cell(TupleCellSpec(table, field)
59      void append_cell(const char *alias, const AggrOp aggr = AGGR_NONE) { append_cell(TupleCellSpec(a
60      int cell_num() const { return static_cast<int>(cells_.size()); }
61
62      const TupleCellSpec &cell_at(int i) const { return cells_[i]; }

```

图 33 存放输出字段信息的类

打开 `TupleCellSpec` 的构造函数，其含有的数据成员有 `table_name`, `field_name` 以及 `alias`，用于表示输出的字段信息。从构造函数来看，该对象只用于描述字段信息而没有任何执行操作，而其中的 `alias` 作为字段的别名将被优先显示，因此我

们可以考虑将聚合函数信息赋值到 **alias** 变量,从而实现输出列名的修改,如图 34 所示。



```
48
49
50 TupleCellSpec::TupleCellSpec(const char *table_name, const char *field_name, const char *alias, const AggrOp aggr)
51 {
52     if (table_name) {
53         table_name_ = table_name;
54     }
55     if (field_name) {
56         field_name_ = field_name;
57     }
58     if (aggr) {
59         aggr_ = aggr;
60     }
61     if (alias) {
62         alias_ = alias;
63     } else {
64         if (table_name_.empty()) {
65             alias_ = field_name_;
66         } else {
67             alias_ = table_name_ + "." + field_name_;
68         }
69
70         if (aggr_ == AggrOp::AGGR_COUNT_ALL) {
71             alias_ = "COUNT(*)";
72         } else if (aggr_ != AggrOp::AGGR_NONE) {
73             std::string aggr_repr;
74             aggr_to_string(aggr_, aggr_repr);
75             alias_ = aggr_repr + "(" + alias_ + ")";
76         }
77     }
78 }
79
80
81 TupleCellSpec::TupleCellSpec(const char *alias, const AggrOp aggr = AggrOp::AGGR_NONE)
82 {
83     if (aggr) {
84         aggr_ = aggr;
85     }
86     if (alias) {
87         alias_ = alias;
88         if (aggr_ == AggrOp::AGGR_COUNT_ALL) {
89             alias_ = "COUNT(*)";
90         } else if (aggr_ != AggrOp::AGGR_NONE) {
91             std::string aggr_repr;
92             aggr_to_string(aggr_, aggr_repr);
93             alias_ = aggr_repr + "(" + alias_ + ")";
94         }
95     }
96 }
97
```

图 34 增加聚合函数的字段输出

其中 **aggr_to_string** 是根据聚合函数类型得到聚合函数对应的字符串的函数,其实现可以自行实现。(比如 **AGGR_MAX** 类型,其显示的字符串则为 **MAX**,具体显示的字符串为什么可以根据本地测试的样例来编写)

此外,在对*的处理中,在 **select_stmt** 的创建函数中将其替换成了所有属性名,则在最后输出时结果就会变成多个属性的形式,而不是 **count(*)**,如图 35 所示。

```
miniob > select count(*) from test;
COUNT(id) | COUNT(value)
2 | 2

miniob > select * from test;
id | value
0 | 12.1
1 | 14.2
```

图 35 输出的不是 count(*)

因此,在 **select_stmt** 的创建函数中,若遇到了 **count(*)**的情况,我们将其标记为 **AGGR_COUNT_ALL**,在最后字段信息的输出时便于进行判断,如图 36 所示。

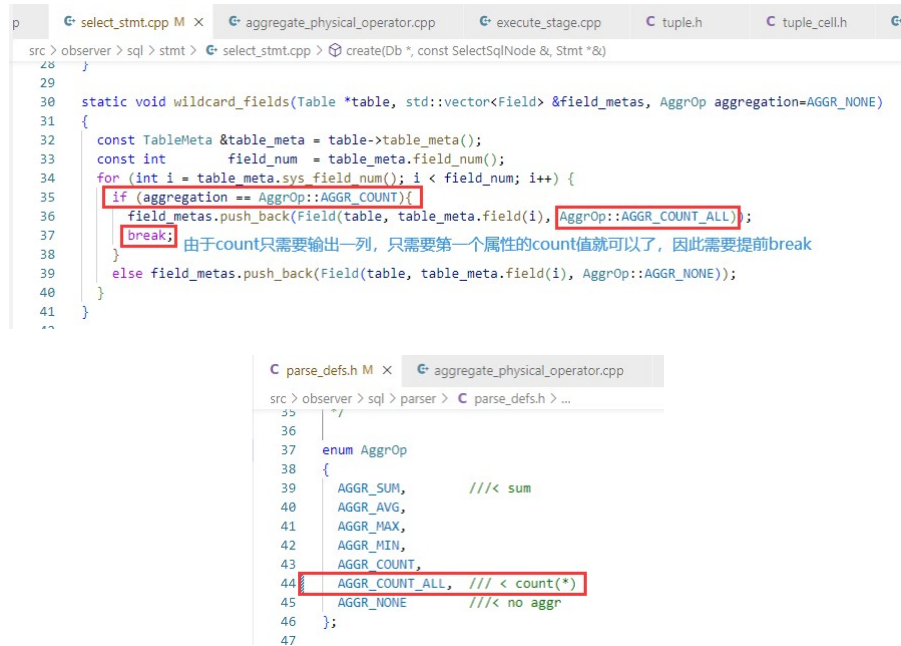


图 36 加入 count(*)的特殊处理

值得注意的是，由于每一列的 count 值一样，且最后只需要输出一个 count 值即可（也就相当于只输出一列的 count），所以不需要把所有的属性名都加进来，只需加一个属性即可。