MiniSQL个人详细报告

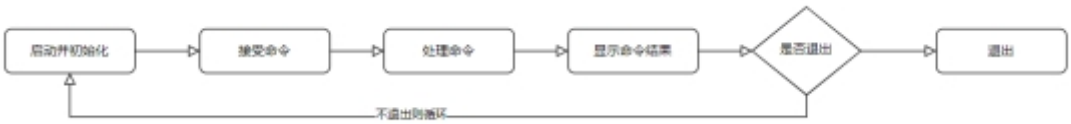
0 分工说明

在MinSQL的项目开发中，我主要负责了#5 SQL EXECUTOR和性能优化的部分

1 SQL EXECUTOR

1.1 模块概述

该模块需要我们理解已有的语法树的数据结构，并解析语法树完成命令执行。即实现一个根据解释器生成的语法树，通过Catalog Manager 提供的信息生成执行计划，并调用相应接口进行执行，最后提供执行上下文ExecuteContext将执行结果返回给上层模块。其流程由下图所示：



执行器通过分析Parser生成的语法树，调用其他模块中的接口，实现用户需求，返回结果信息

1.2 主要功能

分析解释器（Parser）生成的语法树，通过Catalog Manager 提供的信息生成执行计划，并调用Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行，最后通过执行上下文ExecuteContext将执行结果返回给上层模块。

1.3 函数接口

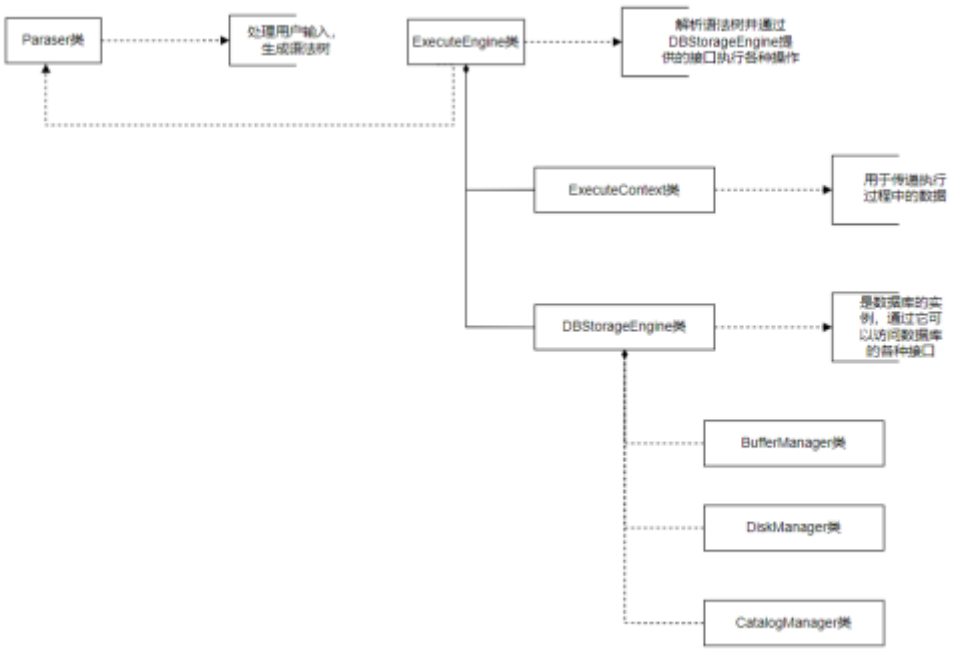
```
//public 对上层屏蔽具体执行细节
dberr_t Execute(pSyntaxNode ast, ExecuteContext *context);

//创建数据库文件
dberr_t ExecuteCreateDatabase(pSyntaxNode ast, ExecuteContext *context);

//删除数据库文件
dberr_t ExecuteDropDatabase(pSyntaxNode ast, ExecuteContext *context);

//删除数据库文件
dberr_t ExecuteShowDatabases(pSyntaxNode ast, ExecuteContext *context);

//切换当前数据库
dberr_t ExecuteUseDatabase(pSyntaxNode ast, ExecuteContext *context);

//显示当前数据库的表
dberr_t ExecuteShowTables(pSyntaxNode ast, ExecuteContext *context);

//新建表
dberr_t ExecuteCreateTable(pSyntaxNode ast, ExecuteContext *context);

//删除表
dberr_t ExecuteDropTable(pSyntaxNode ast, ExecuteContext *context);
```

```
//显示当前数据库的索引
dberr_t ExecuteShowIndexes(pSyntaxNode ast, ExecuteContext *context);

//新建索引
dberr_t ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext *context);

//删除索引
dberr_t ExecuteDropIndex(pSyntaxNode ast, ExecuteContext *context);

//查找操作，支持条件查找
dberr_t ExecuteSelect(pSyntaxNode ast, ExecuteContext *context);

//插入操作
dberr_t ExecuteInsert(pSyntaxNode ast, ExecuteContext *context);

//删除操作，支持条件
dberr_t ExecuteDelete(pSyntaxNode ast, ExecuteContext *context);

//更新操作，支持条件
dberr_t ExecuteUpdate(pSyntaxNode ast, ExecuteContext *context);

//执行文件
dberr_t ExecuteExecfile(pSyntaxNode ast, ExecuteContext *context);

//退出
dberr_t ExecuteQuit(pSyntaxNode ast, ExecuteContext *context);
```

1.4 设计思路

- 初始化：执行器初始化时，会读取目标文件夹下所有数据库文件，并用每个文件的路径初始化DBStorageEngine对象，DBStorageEngine又会初始化Catalog Manager和Buffer Pool Manager，初始化完成时，执行器会获取到不同数据库下的所有表和索引信息。
- **处理语法树：执行器根据语法树的第一个节点类型使执行器跳转到不同的处理函数，在不同处理函数中对语法树进行进一步的处理，处理的主要思路是，先获取已知位置的信息。**

1.5 类图



1.6 实现细节

初始化：执行器初始化时，会读取目标文件夹下所有数据库文件，并用每个文件的路径初始化DBStorageEngine对象，DBStorageEngine又会初始化Catalog Manager和Buffer Pool Manager，初始化完成时，执行器会获取到不同数据库下的所有表和索引信息

处理语法树：执行器根据语法树的第一个节点类型使行器跳转到不同的处理函数，在不同处理函数中对语法树进行进一步的处理，处理的主要思路是，先获取已知位置的信息，比如表名，一般就在当前节点的子节点中，再递归遍历语法树，根据语法树的不同节点类型触发不同事件，不同事件之间信息的传递利用Execute Context进行

```
struct ExecuteContext {
  char* DBname{nullptr};
  char* table_name{nullptr};
  char* index_name{nullptr};
  string index_type{"bptree"};
  struct timeval start;//计时器
  struct timeval end{};
  int col_id{0};//创建新表时待分配col_id
  uint rows_num{0};//计数器
  bool print_flag{0};//PrintInfo两种不同的输出格式
  bool flag_quit_{false};
  bool dont_print{false};//执行文件时，中途不输出
  bool has_condition{false};
  bool is_and{false};//是否为and，是则在最后一个result_index_container的基础上更新，否或者result_index_container为空创建新container
  bool index_only{true};
  bool has_end{false};
  vector<uint32_t> key_map;//主键列号，添加索引的列号，select的列号
  vector<uint32_t> unique_col;//unique列的列号
  vector<Column *> columns;//表的模式
  vector<string> condition_colname;
  vector<vector<RowId>> result_rids_container;
  vector<vector<uint32_t>> result_index_container;//存放临时结果
  unordered_map<string, IndexInfo*> index_on_one_key;
```

```
  unordered_map<int64_t, uint32_t> rowid2index_map;//索引查找时用来将RowId转换成对应行的容器下标
  unordered_map<uint32_t, int64_t> index2rowid_map;
  unordered_map<string, vector<string>> value_i_of_col_;//某一列的所有值
  unordered_map<uint, Field> update_field_map;
  Transaction *txn_{nullptr};
};
```

递归函数

```
void ExecuteEngine::Next(pSyntaxNode ast, ExecuteContext *context) {
  switch (ast->type_) {
    case kNodeColumnDefinition:
      ExecuteNodeColumnDefinition(ast, context);
      break;
    case kNodeColumnList:
      ExecuteNodeColumnList(ast, context);
      break;
    case kNodeConnector:
      ExecuteNodeConnector(ast, context);
      break;
    case kNodeCompareOperator:
      ExecuteNodeCompareOperator(ast, context);
      break;
    case kNodeUpdateValue:
      ExecuteNodeUpdateValue(ast, context);
      break;
    default: {
      if (ast->child_ != nullptr) Next(ast->child_, context);
      if (ast->next_ != nullptr) Next(ast->next_, context);
      break;
    }
  }
}
```

多条件删改查的处理:

处理条件时，条件节点会将符合条件的元组放在结果容器容器(vector<vector<uint32_t>> result_index_container;)中，如果当前节点为and则将is_and置为true，如果当前节点为or则将is_and置为false，如果 is_and为真，当前条件节点在result_index_container.back()基础上进行条件筛选，如果is_and为假，则在当前条件节点遍历整个表并将结果容器插入result_index_container，这样就实现了多条件查询

```
void ExecuteEngine::ExecuteNodeConnector(pSyntaxNode ast, ExecuteContext *context){
  string connector = ast->val_;
  if (connector == "and") {
    if (ast->child_->type_ != kNodeConnector &&
        context->index_on_one_key.find(ast->child_->child_->val_) !=
        context->index_on_one_key.end()) {
      Next(ast->child_->next_, context);
      context->is_and = true;
      Next(ast->child_, context);
    } else {
      Next(ast->child_, context);
      context->is_and = true;
```

```
      Next(ast->child_->next_, context);
    }
  } else if (connector == "or") {
    Next(ast->child_, context);
    context->is_and = false;
    Next(ast->child_->next_, context);
  }
}
```

2 性能优化

1.DiskManager中保存Bitmap对象的指针数组，构造时把所有的 bitmap读进内存

这个比较容易想到，因为Bitmap在分配新页或者检查页是否被分配时会经常被用到，在FetchPage每次都会用IsPageFree检查页号是否合法，而由于在程序中FetchPage使用得非常频繁，不把bitmap读进内存会导致程序疯狂io磁盘[ac01]，一个直观的例子就是，在不把bitmap读进内存而我的FetchPage又会每次做检查的情况下，我插入10000条记录需要2分钟，而把检查去掉或者把bitmap读进内存速度提高到了1w条/s

2.每次往表中插数据时，会先遍历一次表查找能放下数据的表，然后再遍历表中的slot找到空槽，我想尽量避免遍历

这里可以维护两个结构，一个是`vector> free_space_heap;//剩余空间和页号pair容器`每次插入时直接从里面弹出一个剩余空间最大的表页页号，另一个是`unordered_map >reusable_slot_map;//每页可复用槽号map`维护每页可复用槽号，也是直接定位避免遍历，而且维护成本低廉，完成这个优化后，我的程序插入速度就提高到了14w条/s了，效果非常显著

3.还有LRUReplacer，我最开始的实现是基于std::list的，因为它支持头删尾插，但它的查找效率是非常低下的

于是我将list换成了`deque>`，其中的bool类型标志着它有没有被pin，还维护了一个`unordered_map>::iterator>`，Pin时通过它直接定位到要pin的`frame_id`在deque中的位置，然后将bool变量置为false，并删除`unordered_map`中的记录，Unpin时先将记录插入deque尾，再将deque最后一个元素迭代器插入`unordered_map`，Victim函数会循环`pop_front`直到找到第一个没有被pin的`frame_id`，这个优化让我的索引插入从2w条/s加速到了20w条/s

4.每次对元组进行删除都会去移动表页中部分其他元组的位置，删除n个元组时间复杂度就是O(N^2)的，有优化空间

可以采用lazy delete先做标记并且把要删除的元组记录在`unordered_map > row_will_delete_map;`中，析构时统一删除，这样做有两个好处，一个是通过算法的优化可以将时间复杂度降低到O(N)，另一个是数据的更新操作也比较集中，不过这部分就没有进行测试了，不知道快了多少