5 PLANNER AND EXECUTOR

5.1 实验概述

本实验主要包括Planner和Executor两部分。Planner的主要功能是将解释器（Parser）生成的语法树，改写成数据库可以理解的数据结构。在这个过程中，我们会将所有sql语句中的标识符（Identifier）解析成没有歧义的实体，即各种C++的类，并通过Catalog Manager 提供的信息生成执行计划。Executor遍历查询计划树，将树上的 PlanNode 替换成对应的 Executor，随后调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行，并将执行结果返回给上层模块。

5.2 Parser生成语法树

考虑到同学们尚未接触到编译原理的相关知识，在本实验中，我们已经为同学们设计好MiniSQL中的Parser模块，与Parser模块的相关代码如下：

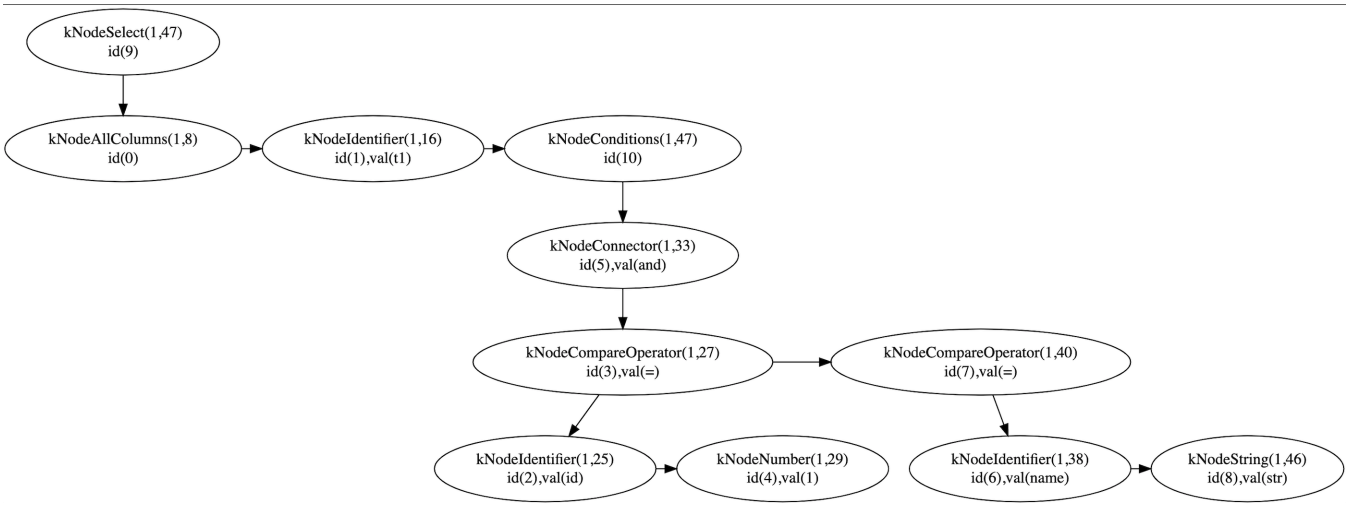
- `src/include/parser/minisql.l`：SQL的词法分析规则；
- `src/include/parser/minisql.y`：SQL的文法分析规则；
- `src/include/parser/minisql_lex.h`：`flex(lex)`根据词法规则自动生成的代码；
- `src/include/parser/minisql_yacc.h`：`bison(yacc)`根据文法规则自动生成的代码；
- `src/include/parser/parser.h`：Parser模块相关的函数定义，供词法分析器和语法分析器调用存储分析结果，同时可供执行器调用获取语法树根结点；
- `src/include/parser/syntax_tree.h`：语法树相关定义，语法树各个结点的类型同样在`SyntaxNodeType`中被定义。

5.2.1 语法树数据结构

以下是语法树（结点）的数据结构定义，每个结点都包含了一个唯一标识符`id_`，唯一标识符在调用`CreateSyntaxNode`函数时生成（框架中已经给出实现）。`type_`表示语法树结点的类型，`line_no_`和`col_no_`表示该语法树结点对应的是SQL语句的第几行第几列，`child_`和`next_`分别表示该结点的子结点和兄弟结点，`val_`用作一些额外信息的存储（如在`kNodeString`类型的结点中，`val_`将用于存储该字符串的字面量）。

```
/**
 * Syntax node definition used in abstract syntax tree.
 */
struct SyntaxNode {
  int id_;    /** node id for allocated syntax node, used for debug */
  SyntaxNodeType type_; /** syntax node type */
  int line_no_; /** line number of this syntax node appears in sql */
  int col_no_;  /** column number of this syntax node appears in sql */
  struct SyntaxNode *child_;  /** children of this syntax node */
  struct SyntaxNode *next_;   /** siblings of this syntax node, linked by a single linked
list */
  char *val_; /** attribute value of this syntax node, use deep copy */
};
typedef struct SyntaxNode *pSyntaxNode;
```

举一个简单的例子，`select * from t1 where id = 1 and name = "str";`这一条SQL语句生成的语法树如下。以根结点为例说明，`kNodeSelect`为结点的类型，`(1,47)`表示该结点在规约（*reduce*，编译原理中的术语）后位于行的第1行第47列（语句末），`id(9)`表示该结点的`id_`为`9`。



5.2.2 Parser支持的SQL语句

Parser模块中目前能够支持以下类型的SQL语句。其中包含了一些在语法定义上正确，但在语义上错误的SQL语句（如Line 8～10）需要同学们在执行器中对这些特殊情况进行处理。此外涉及到事务开启、提交和回滚相关的`begin`、`commit`和`rollback`命令可以不做实现。

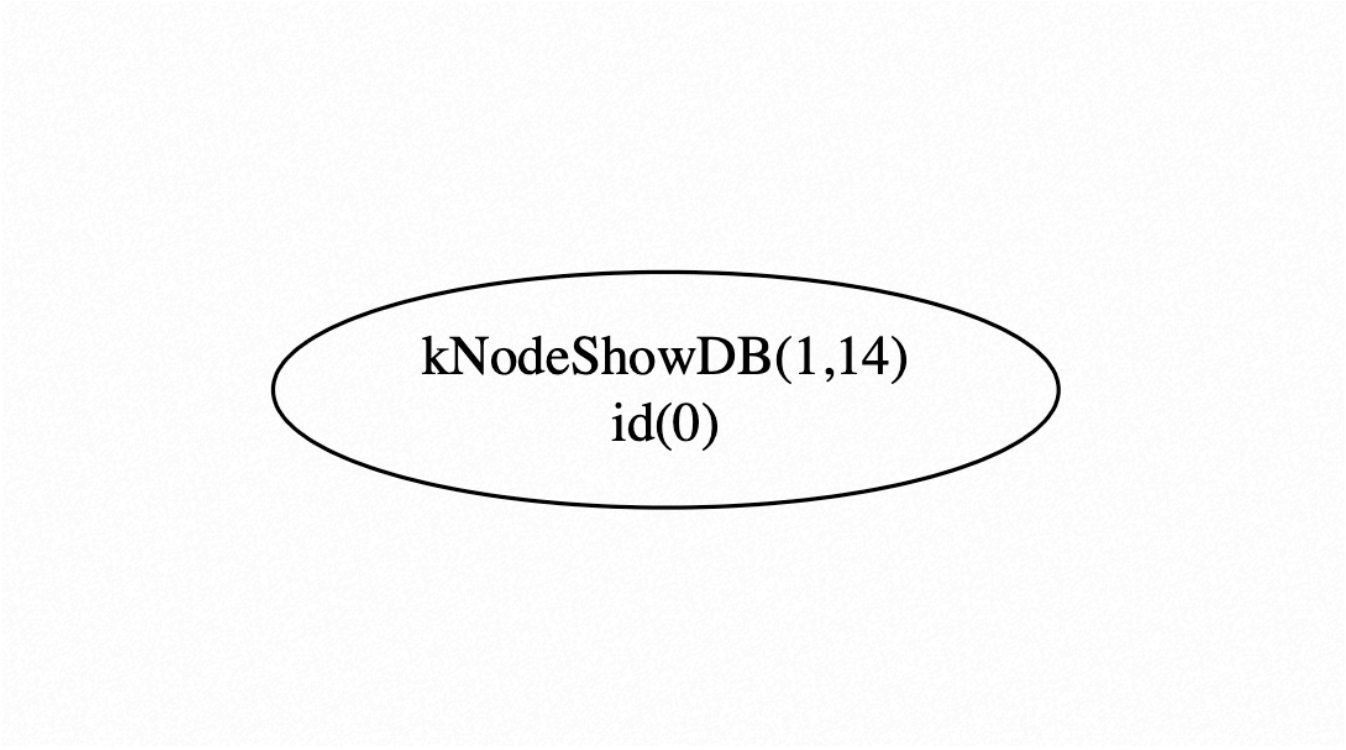
```
create database db0;
drop database db0;
show databases;
use db0;
show tables;
create table t1(a int, b char(20) unique, c float, primary key(a, c));
create table t1(a int, b char(0) unique, c float, primary key(a, c));
create table t1(a int, b char(-5) unique, c float, primary key(a, c));
create table t1(a int, b char(3.69) unique, c float, primary key(a, c));
create table t1(a int, b char(-0.69) unique, c float, primary key(a, c));
create table student(
  sno char(8),
  sage int,
  sab float unique,
  primary key (sno, sab)
);
drop table t1;
create index idx1 on t1(a, b);
-- "btree" can be replaced with other index types
create index idx1 on t1(a, b) using btree;
drop index idx1;
show indexes;
select * from t1;
select id, name from t1;
select * from t1 where id = 1;
-- note: use left association
select * from t1 where id = 1 and name = "str";
select * from t1 where id = 1 and name = "str" or age is null and bb not null;
```

```
insert into t1 values(1, "aaa", null, 2.33);
delete from t1;
delete from t1 where id = 1 and amount = 2.33;
update t1 set c = 3;
update t1 set a = 1, b = "ccc" where b = 2.33;
begin;
commit;
rollback;
quit;
execfile "a.txt";
```

在Parser模块调用`yyparse()`（一个示例在`src/main.cpp`中）完成SQL语句解析后，将会得到语法树的根结点`pSyntaxNode`。将语法树根结点传入`ExecuteEngine`（定义于`src/include/executor/execute_engine.h`）后，`ExecuteEngine`将会根据语法树根结点的类型，决定是否需要传入`Planner`生成执行计划。

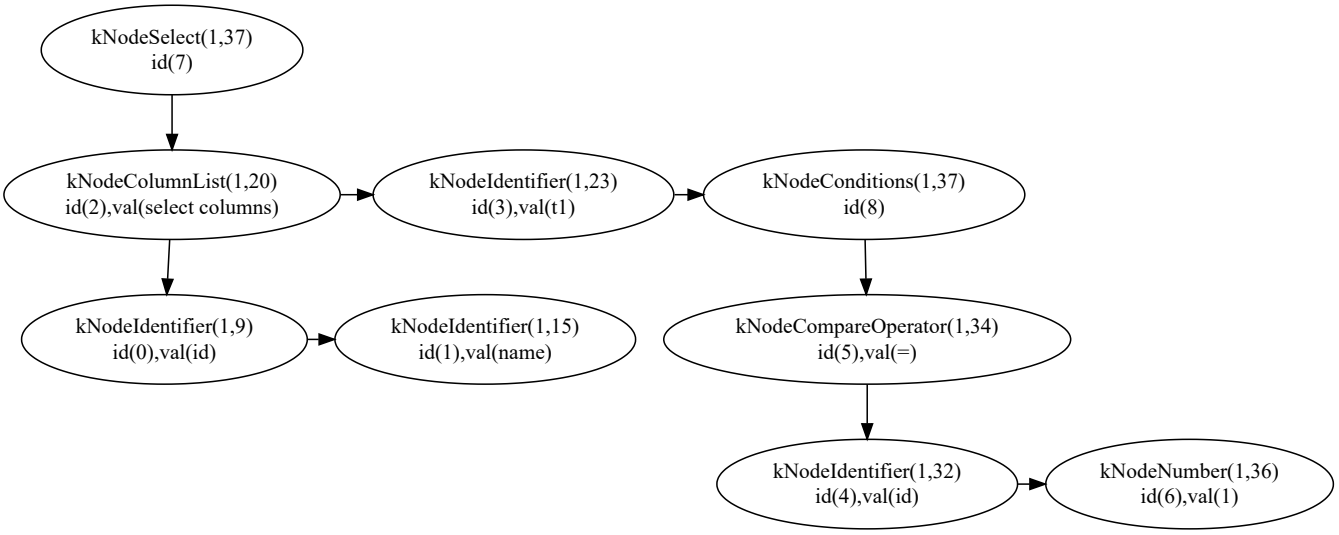
5.3 Planner生成查询计划

对于简单的语句例如show databases，drop table 等，生成的语法树也非常简单。以show databases为例，对应的语法树只有单节点kNodeShowDB，表示展示所有数据库。此时无需传入Planner生成执行计划，我们直接调用对应的执行函数执行即可。



而复杂的语句例如select，insert，update，deleete，生成的语法树也较为复杂。以一条select的sql为例，生成的对应语法树如下：其中 kNodeSelect标识了语句的类型，kNodeColumnList标识了有哪些列，kNodeIdentifier标识了是哪张表，kNodeConditions标识了where语句的条件。

```
select id, name from t1 where id = 1;
```



对于复杂的语句，生成的语法树需传入Planner生成执行计划，并交由Executor进行执行。Planner需要先遍历语法树，调用Catalog Manager 检查语法树中的信息是否正确，如表、列是否存在，谓词的值类型是否与column类型对应等等，随后将这些词语抽象成相应的表达式（表达式在`src/include/planner/expressions/`），即可以理解的各种 c++ 类。解析完成后，Planner根据改写语法树后生成的可以理解的Statement结构，生成对应的Plannode，并将Planndoe交由executor进行执行。该模块相关的代码如下：

- `src/include/planner/statement/abstract_statement.h`
- `src/include/planner/statement/select_statement.h`
- `src/include/planner/statement/insert_statement.h`
- `src/include/planner/statement/delete_statement.h`
- `src/include/planner/statement/update_statement.h`

`Statement`中的函数`SyntaxTree2Statement`将解析语法树，并将各种Identifier转化为可以理解的表达式，存储在Statement结构中。Planner再根据Statement，生成对应的执行计划，相关代码如下：

- `src/include/executor/plans/abstract_plan.h`
- `src/include/executor/plans/delete_plan.h`
- `src/include/executor/plans/insert_plan.h`
- `src/include/executor/plans/seq_scan_plan.h`
- `src/include/executor/plans/update_plan.h`
- `src/include/executor/plans/value_plan.h`

本模块中你不用实现任何代码。

Tips：在成熟的数据库中，Planner一般和优化器Optimizer一起，称为查询优化器。通常，查询优化器会通过如下三个典型组件协同来完成查询优化。优化后，能将原本根据语法树直接生成的查询计划改写成效率更高的查询计划，例如经典的join order问题。

- **Plan space enumeration:** 根据一系列的等价变换规则，生成与查询等价的多个执行计划；
- **cardinality estimation:** 根据查询表的分布情况，估计查询执行过程中的数据量/数据分布等；
- **cost model:** 根据执行计划以及数据库内部的状态，计算按照各个执行计划执行所需要的代价。

5.4 Executor执行查询计划

在拿到 Planner 生成的具体的查询计划后，就可以生成真正执行查询计划的一系列算子了。生成算子的步骤很简单，遍历查询计划树，将树上的 PlanNode 替换成对应的 Executor。算子的执行模型也大致分为三种：

1. Iterator Model，即经典的火山模型。执行引擎会将整个 SQL 构建成一个 Operator 树，查询树自顶向下的调用接口，数据则自底向上的被拉取处理。每一种操作会抽象为一个 Operator，每个算子都有 Init() 和 Next() 两个方法。Init() 对算子进行初始化工作。Next() 则是向下层算子请求下一条数据。当 Next() 返回 false 时，则代表下层算子已经没有剩余数据，迭代结束。
2. 该方法的优点是其计算模型简单直接，通过把不同物理算子抽象成一个个迭代器。每一个算子只关心自己内部的逻辑即可，让各个算子之间的耦合性降低，从而比较容易写出一个逻辑正确的执行引擎。
3. 缺点是火山模型一次调用请求一条数据，占用内存较小，但函数调用开销大，特别是虚函数调用造成 cache miss 等问题。同时，逐行地获取数据会带来过多的 I/O，对缓存也不友好。
4. Materialization Model，算子计算出所有结果后一起返回。这种模型的弊端显而易见，当数据量较大时，内存占用很高。但该模型减少了函数调用的开销。比较适合查询数据量较小的 OLTP workloads。
5. Vectorization Model. 对上面两种模型的中和，输入和输出都以Batch为单位。在Batch的处理模式下，计算过程还可以使用SIMD指令进行加速。目前比较先进的 OLAP 数据库都采用这种模型。

本任务采用的是最经典的 Iterator Model。在本次任务中，我们将实现5个算子，分别是select，Index Select，insert，update，delete。 对于每个算子，都实现了 Init 和 Next 方法。 Init 方法初始化运算符的内部状态，Next 方法提供迭代器接口，并在每次调用时返回一个元组和相应的 RID。对于每个算子，我们假设它在单线程上下文中运行，并不需要考虑多线程的情况。每个算子都可以通过访问 `ExecuteContext` 来实现表的修改，例如插入、更新和删除。 为了使表索引与底层表保持一致，插入删除时还需要更新索引。

最后，我们在 `test/execution/executor_test.cpp` 中提供了一些执行器的测试样例。这些样例非常简单，但可以通过它们了解PlanNode是如何构建的，以及算子是如何执行PlanNode并得到期望的结果。

5.3.1 SeqScan

SeqScanExecutor 对表执行一次顺序扫描，一次Next()方法返回一个符合谓词条件的行。顺序扫描的表名和谓词（Predicate）由SeqScanPlanNode 指定。

提示：遍历表时可使用 TableIterator 对象，并了解清楚 ++iter 和 iter++ 在底层是怎样实现的。

提示：如果Scan存在谓词，需要通过调用谓词表达式的Evaluate方法来判断该行是否满足条件。建议了解清楚不同表达式的AbstractExpression::Evaluate方法是如何实现的。目前底层没有Bool类型，所以Evaluate方法返回的结果可以和Field(kTypeInt, 1)进行比较，相等则为True。

提示：顺序扫描的输出是每个匹配行及其标识符（RID）的副本。

5.3.2 IndexScan

IndexScanExecutor 对表执行一次带索引的扫描，一次Next()方法返回一个符合谓词条件的行。为简单起见，IndexScan仅支持单列索引。当Planner检测到select的谓词中的列上存在索引，而且索引只包含该列时，会生成IndexScanPlan，其他情况则生成SeqScanPlan。同时，为简单起见，IndexScan不支持谓词中存在or的情况（例如a=1 or b=2，此时a列存在索引，b列不存在索引，我们仍需顺序扫描表来找出符合b=2条件的列）。因此，在IndexScanExecutor中你只需要考虑单列索引，并且不用考虑谓词中存在or的情况。

提示：indexes_中存放了所有的索引，我们需要求出满足每个索引的rowid集合，并对这些集合取交集（因为只用考虑and情况）。之后根据rowid，去取相应的记录。取交集可以用库函数set_intersection方法。

提示：need_filter_表示是否谓词中所有的列上都有索引。如果只有部分列上有索引，那么我们用索引筛选出符合条件的记录后，仍需再调用evaluate方法判断是否符合条件。

5.3.3 Insert

InsertExecutor 将行插入表中并更新索引。目前，Parser只支持直接插入`insert into t1 values(1, "aaa", null, 2.33);`，不支持 INSERT INTO .. SELECT ...的语法。要插入的值通过ValueExecutor生成对应的行，随后被拉取到InsertExecutor中。

提示：注意unique对插入的影响。因为建表时会对所有unique列建立索引，所以查询插入值是否已经存在可以走索引。

提示：由于可以对不同的字段建立 index，一个 table 可能对应多个 index，所有的 index 都需要更新。

提示：返回空的 tuple ，用于统计 table 中有多少行受到了影响。

5.3.4 Value

ValueExecutor主要用于`insert into t1 values(1, "aaa", null, 2.33);`语句。插入的值以vector形式存储在ValuesPlanNode中，ValueExecutor调用Next()方法一次返回一个新的行。本算子已经实现。

5.3.5 Update

UpdateExecutor 修改指定表中的现有行并更新其索引。UpdatePlanNode 将有一个 SeqScanPlanNode 作为其子节点，要更新的值通过SeqScanExecutor提供。

提示：Update可理解为先删除旧行，再插入新行。

提示：GenerateUpdatedTuple方法将根据PlanNode中提供的更新属性构造出一个新的元组。

5.3.6 Delete

DeleteExecutor 删除表中符合条件的行。和Update一样，DeletePlanNode 将有一个 SeqScanPlanNode 作为其子节点，要删除的值通过SeqScanExecutor提供。

提示：注意索引的删除。

提示：返回空的 tuple ，用于统计 table 中有多少行受到了影响。

在本节中，你需要实现seqscan，insert，update，delete, indexscan这五个算子。算子的头文件定义如下：

- `src/include/executor/executors/abstract_executor.h`
- `src/include/executor/executors/delete_executor.h`
- `src/include/executor/executors/insert_executor.h`
- `src/include/executor/executors/seq_scan_executor.h`
- `src/include/executor/executors/index_scan_executor.h`
- `src/include/executor/executors/values_executor.h`

此外，你还需要实现`src/include/executor/execute_engine.h`中的创建删除查询数据库、数据表、索引等函数。它们对应的语法树较为简单，因此不用通过Planner生成查询计划。它们被声明为`private`类型的成员，所有的执行过程对上层模块隐藏，上层模块只需要调用`ExecuteEngine::execute()`并传入语法树结点即可无感知地获取到执行结果。

- `ExecuteEngine::ExecuteCreateDatabase(*ast, *context)`
- `ExecuteEngine::ExecuteDropDatabase(*ast, *context)`
- `ExecuteEngine::ExecuteShowDatabases(*ast, *context)`
- `ExecuteEngine::ExecuteUseDatabase(*ast, *context)`
- `ExecuteEngine::ExecuteShowTables(*ast, *context)`
- `ExecuteEngine::ExecuteCreateTable(*ast, *context)`
- `ExecuteEngine::ExecuteDropTable(*ast, *context)`
- `ExecuteEngine::ExecuteShowIndexes(*ast, *context)`
- `ExecuteEngine::ExecuteCreateIndex(*ast, *context)`
- `ExecuteEngine::ExecuteDropIndex(*ast, *context)`
- `ExecuteEngine::ExecuteExecfile(*ast, *context)`
- `ExecuteEngine::ExecuteQuit(*ast, *context)`
- `ExecuteEngine::ExecuteTrxBegin(*ast, *context)`：事务相关，可不实现
- `ExecuteEngine::ExecuteTrxCommit(*ast, *context)`：事务相关，可不实现
- `ExecuteEngine::ExecuteTrxRollback(*ast, *context)`：事务相关，可不实现

5.4 模块相关代码

- `src/main.cpp`
- `src/include/executor/execute_engine.h`
- `src/executor/execute_engine.cpp`
- `src/include/executor/executors/abstract_executor.h`
- `src/include/executor/executors/delete_executor.h`
- `src/include/executor/executors/insert_executor.h`
- `src/include/executor/executors/seq_scan_executor.h`
- `src/include/executor/executors/index_scan_executor.h`
- `src/include/executor/executors/update_executor.h`

5.5 开发提示

1. 整个MiniSQL项目推荐在**夏学期第7周**前完成；
2. 框架中已经给出了语法树的`PrintTree()`方法，它能够打印语法树中的每一个结点（输出DOT格式），具体用法和之前的B+树打印类似，输出的结果放在可视化界面中可以用作调试。此外也可以使用GDB或IDE自带的调试工具在完成SQL语法分析后得到语法树的语句打上断点以进行调试。

3. 如果需要更改语法和文法以支持新的SQL命令，可以在学习LEX和YACC的相关知识后，修改`src/include/parser/minisql.l`和`src/include/parser/minisql.y`文件，然后执行`src/include/parser/compile.sh`脚本（它会自动生成对应的`lex`和`yacc`代码并移动到工程的指定目录下），最后需要重新执行`cmake ..`完成更新；
4. 在`test/execution/executor_test.cpp`中提供了一些执行器的测试样例。在executor实现之后，可以在`main.cpp`中手动输入SQL命令观察结果。

5.6 诚信守则

1. 请勿从其它组或在网络上找到的其它来源中复制源代码，一经发现抄袭，成绩为`0`；
2. 请勿将代码发布到公共Github存储库上。

5.7 评论

请问 IndexScan里这两个点啥意思呀？难道不应该是利用index执行SeqScan的功能就可以了吗？
"提示：indexes*中存放了所有的索引，我们需要求出满足每个索引的rowid集合，并对这些集合取交集（因为只用考虑and情况）。之后根据rowid，去取相应的记录。取交集可以用库函数set_intersection方法。*
" *提示：need_filter*表示是否谓词中所有的列上都有索引。如果只有部分列上有索引，那么我们用索引筛选出符合条件的记录后，仍需再调用evaluate方法判断是否符合条件

以a=1 and b=2的情况为例。当a、b列都建立了索引，此时need_filter*为false，意味着只通过索引就可以获得最终的结果集，不需要额外的过滤。此时根据索引获得满足a=1的rowid集合，b=2的rowid集合，并取交集，然后next依次输出。当只有a列建立了索引，此时need_filter*为true，我们根据索引获得满足a=1的rowid集合，然后还需要通过evaluate方法判断是否满足b=2的条件的再输出。

winson_Liu2023-05-26 11:21IP 属地浙江

引用原文：提示：返回空的 tuple ，用于统计 table 中有多少行受到了影响。

这是什么意思？能详细说一下吗

意思是Next方法里的参数Row *row, RowId *rid，不需要进行赋值，这样Next调用结束后row还是空的

引用原文：同时，为简单起见，IndexScan不支持谓词中存在or的情况（例如a=1 or b=2，此时a列存在索引，b列不存在索引，我们仍需顺序扫描表来找出符合b=2条件的列）

这里的or和and的意思是什么，能举个具体的例子吗

就是谓词里有and和or的条件啊，具体例子可参考上面评论

winson_Liu2023-05-26 11:25IP 属地浙江

引用原文：need_filter_表示是否谓词中所有的列上都有索引

need_filter 在plan_node中已经被定义为true了，这个need_filter到底是什么意思。

这只是一个默认初值，具体看PlanSelect函数，以及上面评论的意思。need_filter_为false，意味着条件中所有列都建有索引，只通过索引就可以获得最终的结果集，不需要再调用evaluate方法进行过滤

winson_Liu2023-05-26 21:56IP 属地浙江

引用原文：要更新的值通过SeqScanExecutor提供。

请问这是怎么提供的?

前面说了，UpdatePlanNode 将有一个 SeqScanPlanNode 作为其子节点。所以更新时，会先通过child_executor_->Next()方法，获得要更新的row

winson_Liu2023-05-27 13:50IP 属地浙江

引用原文：set_intersection

这个要求容器内的元素是有序的，RowId里面并没有比较运算符，怎么使用？

自己定义一个就行了……set_intersection支持传入自定义的比较函数

winson_Liu2023-05-29 19:29IP 属地浙江

感谢感谢！

icey2023-06-09 02:41IP 属地中国台湾

引用原文：顺序扫描的表名和谓词（Predicate）由SeqScanPlanNode 指定。

请问如何通过表明找到对应的表呢？

通过exec_ctx_获得Catalog，再调用GetTable()函数

icey2023-06-09 02:49IP 属地中国台湾

引用原文：要插入的值通过ValueExecutor生成对应的行，随后被拉取到InsertExecutor中。

请问什么叫做“被拉取到”?

即调用child_executor_的Next方法

AutonomierLong2024-05-27 22:01IP 属地浙江

引用原文：在本节中，你需要实现seqscan，insert，update，delete, indexscan这五个算子。算子的头文件定义如下：

所以今年不用实现这些了吗, 为什么代码都已经给出了?

今年增加模块6，和bonus模块7，所以减轻负担，5只用实现几个建表之类的函数