**Query Execution(SQL执行器)**

现在我们实现一个执行器，但不解析SQL。

现在我们完成数据库系统的查询执行(query execution)功能。我们将实现负责获取查询计划节点(query plan nodes)并执行他们的执行器(executors)，该执行器需要执行以下操作：

存取方法：Sequential Scan（顺序扫描）

修改类型：Insert, Update, Delete

其他：Nested Loop Join（循环嵌套join）, Hash Join, Aggregation, Limit, Distinct

这时DBMS还不支持SQL，所以我们的实现将直接在手工编写的查询计划上运行。

我们将使用迭代器查询执行模型（又称火山模型,iterator query processing model,Volcano model）。在该模型中，每个查询计划执行器都实现了一个Next函数。当DBMS调用执行器的Next函数时，执行器返回单个元组(记录)或返回一个表示没有更多元组(记录)的标识符。通过这种方法，每个执行器实现一个循环，该循环不断调用其子对象的Next函数来检索元组(记录)并逐个处理他们。

在DBMS的迭代器模型实现中，每个执行器的Next函数除了返回一个元组(记录)外，还返回一个记录标识符（Record Identifier,RID），记录标识符是记录在它所属的表中的唯一标识。

**一．执行器说明**

我们将实现九种执行器，即:sequential scan,insert, update, delete,nested loop Join, hash Join, aggregation, limit, distinct。对于每种查询计划运算符类型，都有一个相应的执行器对象来实现Init和Next方法。Init方法初始化操作符的内部状态（如对相应的表进行检索和扫描）。Next方法提供迭代器的接口，该接口在每次调用时返回一个记录和对应的RID（也可能是执行器已经结束的标志）。

我们实现的执行器的头文件如下：

**src/include/execution/executors/seq\_scan\_executor.h**

**src/include/execution/executors/insert\_executor.h**

**src/include/execution/executors/update\_executor.h**

**src/include/execution/executors/delete\_executor.h**

**src/include/execution/executors/nested\_loop\_join\_executor.h**

**src/include/execution/executors/hash\_join\_executor.h**

**src/include/execution/executors/aggregation\_executor.h**

**src/include/execution/executors/limit\_executor.h**

**src/include/execution/executors/distinct\_executor.h**

每个执行器负责处理单个的计划节点(plan node)类型。Plan Node是组成查询计划的各个元素。每个Plan Node可以定义他所代表的操作符的特定信息。如：顺序扫描(sequential scan)的Plan Node必须为执行扫描的表定义标识符，而Limit的Plan Node不需要此信息。这些Plan Node已经在如下头文件中定义：

**src/include/execution/plans/seq\_scan\_plan.h**

**src/include/execution/plans/insert\_plan.h**

**src/include/execution/plans/update\_plan.h**

**src/include/execution/plans/delete\_plan.h**

**src/include/execution/plans/nested\_loop\_join\_plan.h**

**src/include/execution/plans/hash\_join\_plan.h**

**src/include/execution/plans/aggregation\_plan.h**

**src/include/execution/plans/limit\_plan.h**

**src/include/execution/plans/distinct\_plan.h**

这些Plan Node已经拥有了实现所需执行器的全部数据和功能，我们不用修改它们。

执行器从他们的子对象（可能是多个）中得到记录，并将记录交给父对象。它们可能依赖于其子对象的顺序，我们下文将会介绍。

我们假设执行器在单线程的上下文中执行。当多个线程并发执行时，我们暂时未采取额外的措施来保证执行器的正确执行。

我们将提供ExecutionEngine这个Helper类。它将输入的查询计划(query plan)转换为查询执行器(query executor)并执行它，直到生成所有结果。我们将修改ExecutionEngine来捕获由于ExecutorFactory类描述了如何在查询执行运行时创建执行器。此外，每个执行器都可以访问它执行的ExecutorContext。

其中一些执行器会对表进行一些修改（插入insert、更新update、删除delete）。为了保持与底层表一致的表索引，这些执行器还需要更新被修改的表的所有索引。我们将使用之前实现的可扩展哈希表作为所有索引的底层数据结构。

另外，在test/execution/executor\_test.cpp文件中，提供了一些测试。

**1.Sequential Scan**

SeqScanExecutor遍历表并返回它的记录。顺序扫描(sequential scan)由SeqScanPlanNode指定，该Plan Node指定被扫描的表，它还包含一个predicate，如果一条记录不满足predicate，则扫描不会提供该记录。

注意，使用TableIterator对象时要注意前增运算符和后增运算符的区别，++iter和iter++之间的切换会得到奇怪的输出。

注意：我们需要在顺序扫描的Plan Node中使用predicate。特别是，注意AbstractExpression::Evaluate，它会返回一个值，在该值上可以调用GetAs<bool>()来将结果作为bool类型来使用。

注意：顺序扫描的结果是每条匹配记录及其源记录标识符(RID,record identifier)。

**2.Insert**

InsertExecutor将记录插入表中并更新索引。我们的执行器需要支持两种插入类型。

第一种插入操作中将要被插入的值直接嵌入到Plan Node中，我们称之为原始插入：INSERT INTO ... VALUES ...

第二种插入操作中，要被插入的值来自于**子执行器**。例如，我们现在有一个InserPlanNode，其中SeqScanPlanNode作为子执行器来实现插入操作:INSERT INTO .. SELECT ...

假设InsertExecutor始终位于其查询计划的根目录下，InsertExecutor不应该修改result set。

注意：在执行器初始化的时候需要查找插入目标的表的信息。有关访问catalog的更多信息，参见后文System Catalog部分。

注意：我们需要更新被插入记录的表的所有索引。更多信息参考后文Index Updates部分。

注意：我们需要使用TableHeap类来执行表修改。

**3.Update**

UpdateExecutor修改指定表中的现有记录并更新其索引。子执行器将提供Update执行器需要修改的记录及其RID。

和InsertExecutor不同，UpdateExecutor总是从子执行器中提取记录并更新。例如，UpdatePlanNode将有一个SeqcScanPlanNode作为其子节点。

我们可以假定UpdateExecutor始终位于其查询计划的根目录下，UpdateExecutor不应该修改result set。

注意：我们提供GenerateUpdatedTuple，它基于Plan Node中提供的Update列为您构建一条Updated的记录。

注意：在执行器初始化的时候需要查找插入目标的表信息。有关访问catalog的更多信息，参见后文System Catalog部分。

注意：我们需要更新有关被插入的记录的所有索引。更多信息参考后文Index Updates部分。

提示：我们需要使用TableHeap类来执行表修改。

**4.Delete**

DeleteExecutor从表中删除记录，并从所有表的索引中删除与其有关的项。和Update一样，要删除的记录也从子执行器，如SeqScanExecutor中得到。

我们可以假定DeleteExecutor始终位于其查询计划的根目录下，DeleteExecutor不应该修改result set。

注意：我们只需要从子执行器中得到RID然后调用TableHeap::MarkDelete()即可有效的删除记录。所有删除将在事务被提交时应用。

注意：我们需要更新有关被删除的记录的所有索引。更多信息参考后文Index Updates部分。

**5.Nested Loop Join**

NestedLoopJoinExecutor实现了一个基本的嵌套循环联接，它将它的两个子执行器中的记录组合在一起。

该执行器实现了**简单嵌套循环连接**算法(参考数据库系统概念477页)。也就是说，对于联接的外部表中的每行记录，应考虑内部表中的每行记录。如果满足联接谓词，则应返回记录。

为了了解逻辑运算符的不同物理实现的优缺点，NestedLoopJoinExecutor必须实现简单嵌套循环联接，而不是使用我们后面实现的Hash Join执行器。这样，我们可以对比Nested Loop Join和Hash Join之间的性能。

注意，我们需要使用嵌套循环联接的Plan Node中的predicate(谓词)。特别是，注意处理左记录和右记录以及它们模式的AbstractExpression::Evaluate，它会返回一个值，在该值上可以调用GetAs<bool>()来将结果作为bool类型来使用。

**6.Hash Join**

HashJoinExecutor实现了一个哈希连接(数据库系统概念482页)操作，该操作将来自其两个子执行器的记录组合在一起。

顾名思义，哈希连接是在哈希表的帮助下实现的。出于简化的目的，我们假设哈希表完全适合内存。这意味着我们不必实现将build-side表临时划出的部分转移到硬盘的操作。

注意：我们的实现应该正确处理多个记录（在连接的任一侧）共享一个common join key的情况。

注意：HashJoinPlanNode定义了GetLeftJoinKey()和GetRightJoinKey()两个成员函数。我们应该使用这些访问器返回的表达式来分别为连接的左侧和右侧构造join key。

注意：我们需要一种方法来散列具有多个列的记录，以便构造其Unique Key。参考AggregationExecutor中的SimpleAggregationHashTable实现此功能。

注意：在查询计划的上下文中，hash join的生成端(build side)是一个pipeline breaker。这可能会影响我们使用HashJoinExecutor::Init()和HashJoinExecutor::Next()的方式。特别的，考虑是在HashJoinExecutor::Init()还是HashJoinExecutor::Next()中执行连接的构建阶段。

**7.Aggregation**

此执行器将来自单个子执行器的多条记录的结果合并到一条记录中(数据库系统概念489)。此执行器需要实现Count，Sum，Min和Max。AggregationExecutor还必须支持Group By和Having子句。

实现聚合(**Aggregation**)的常见策略是使用哈希表，其中我们简化了一些步骤，即聚合哈希表完全适合内存，这意味着我们不必实现哈希聚合的两阶段（Partition，Rehash）策略，而是可以假设所有聚合结果都可以保留在内存的哈希表中。

此外，我们还提供了SimpleAggregationHashTable数据结构。该结构公开了内存中的哈希表（std::unordered\_map），但还提供了一个用于计算聚合的结构。这个类还公开了SimpleAggregationHashTable::Iterator的类型，该类型可用于遍历哈希表。

注意：您需要聚合结果并将Having用于约束。特别的，看一下AbstractExpression::EvaluateAggregate，它处理不同类型表达式的聚合计算，它会返回一个值，在该值上可以调用GetAs<bool>()来将结果作为bool类型来使用。

注意：在查询计划的上下文中，聚合是一个pipeline breaker。这可能会影响我们使用AggregationExecutor::Init()和AggregationExecutor::Next()的方式。特别的，考虑是在AggregationExecutor::Init()还是AggregationExecutor::Next()中执行聚合的构建阶段。

**8.Limit**

LimitExecutor约束其子执行器输出的记录行数。如果其子执行器生成的记录数小于LimitPlanNode中指定的限制，则此执行器无效，并返回其接收到的所有记录。

**9.Distinct**

DistinctExecutor消除从其子执行器得到的重复记录(数据库系统概念486)。在唯一确定的情况下，DistinctExecutor应该考虑输入元组的所有列。

与聚合一样，distinct运算符通常在哈希表的帮助下实现（这是一个哈希distinct操作）。这是您在本实验中将会使用到的方法。和AggregationExecutor与HashJoinExecutor一样，我们可以假设用于实现DistinctExecutor的哈希表完全适合内存。

注意：为了构造一个唯一的键，您需要一种方法来散列可能包含很多列的记录。参考AggregationExecutor中的SimpleAggregationHashTable是如何实现此功能的。

**实现细节说明：**

1. 测试代码利用table\_generator生成表，predicate是表示条件的谓词，Schema是要返回的列的模式，包括要返回的列的列数，列的表达式等(以column数组的形式存放)，Evaluate函数会根据Schema(表达式)返回表达式运算后的值。

2.std::move是将对象的状态或者所有权从一个对象转移到另一个对象，只是转移，没有内存的搬迁或者内存拷贝所以可以提高利用效率，改善性能。

3.注意abstract executor和abstract execution，它们是基类，别的executor和execution从它们衍生而来。具体的执行过程为：通过execution engine调用executor，executor是一个工厂类，executor根据plan的类型返回具体的executor指针，然后execution engine根据返回的executor指针执行查询计划。

**二．其他**

**1.System Catalog**

数据库维护一个内部目录，以跟踪有关数据库的元数据。我们将与系统目录交互，以查询有关表、索引及其模式的信息。

整个目录实现都在src/include/catalog.h中。注意成员函数Catalog::GetTable()和Catalog::GetIndex()，在执行器的实现中使用这些函数来查询目录中的表和索引。

**2.Index Updates**

对于表修改执行器（InsertExecutor、UpdateExecutor和DeleteExecutor），需要修改操作所在表的索引，这时Catalog::GetTableIndexes()函数对于查询为特定表定义的所有索引非常有用。一旦为每个表的索引创建了IndexInfo实例，就可以对基础索引结构调用索引修改操作。

注意，我们使用之前实现的可扩展哈希表作为所有索引操作的底层数据结构。