第 9 章 查询处理

9.1 实验介绍

本实验通过阅读和分析 openGauss 数据库服务器中嵌套循环连接算法实现源代码，理解和验证嵌套循环连接算法的实现机制。首先介绍嵌套循环连接算法的原理，通过浏览 ExecNestLoop 函数源代码和相应的流程图，实践嵌套循环连接算法的具体工程实现。在浏览阅读相关结构体源代码之后，以 railway 数据库中的 users 表和 orders 表的连接查询作为示例，通过添加源代码进行宏重定义的方式，尝试输出 ExecNestLoop 函数调试信息。通过配置优化器参数改变查询执行计划，引发 ExecNestLoop 函数执行。最后，通过添加代码，输出嵌套循环算法中进行连接属性值的比较信息，进一步理解循环嵌套连接的查询执行计划中索引扫描与顺序扫描的区别。

本实验的实践内容涉及到数据库服务器的查询执行与查询优化机制，相关 openGauss 源代码关联的细节众多，具有一定的复杂性和挑战性。在添加代码过程中，引入错误（bug）也是不可避免的一件事情。在诸如 openGauss 这样大型、复杂、服务器架构的系统软件中，通过贯彻“编辑——编译——测试——调试”的迭代步骤，采取系统性的调试手段修正错误，可以有效地提升解决复杂系统工程问题的能力并增长相应的系统开发与调试经验。

9.2 实验目的

1. 理解嵌套循环连接算法的原理。
2. 理解 ExecNestLoop 函数代码实现与嵌套循环连接算法的对应关系。
3. 掌握通过宏重定义方式输出 ExecNestLoop 函数调试信息的方法。
4. 掌握通过在 ExecNestLoop 函数添加代码输出嵌套循环算法中字段值的方法。
5. 理解通过 EXPLAIN 语句查看的查询执行计划。
6. 理解通过优化器参数配置改变查询执行计划的方式。
7. 了解与本实验相关的结构体与源代码。

9.3 实验原理

9.3.1 嵌套循环连接算法原理

两个表的连接是数据库查询中比较常用的一种操作，而嵌套循环连接是通过嵌套的循环语句把多个表连接起来的简单算法，称为嵌套循环（nested loop）算法。

嵌套循环连接（Nested Loops Join）是一种使用两层循环（外层循环和内层循环）来实现表连接操作，从而得到连接结果集的表连接方法。即，内层循环遍历次数取决于外层循环所对应的驱动结果集的记录条数，这就是所谓的“嵌套循环”的含义。

“嵌套循环连接算法”在数据库查询中用来执行连接操作。它通过在两个数据表上进行嵌套循环来实现表之间的连接。每次外循环读入一行数据，内循环在另一个数据表中搜索匹配

的行。如果找到匹配，则返回结果，否则继续读入下一行。嵌套循环连接的复杂度为 O(*mn*)，其中 *m* 和 *n* 是两个关系的行数。它的效率在行数很小的情况下很高，如果关系中行数非常大，该算法的效率将降低。因此，当数据量非常大时，通常需要使用其他更高效的算法来代替嵌套循环连接算法。以下是"嵌套循环连接算法"的优点、缺点及适用场景的描述：

- 如果驱动表对应的驱动结果集记录数较少，并且被驱动表的连接列存在唯一性索引或选择性较高的非唯一性索引，使用嵌套循环连接的效率将非常高。但是，如果驱动表对应的驱动结果集记录数很多，即使存在索引，使用嵌套循环连接的效率也不会高。
- 数据量大的表作为嵌套循环连接的驱动表时，要看所在 SQL 语句中是否有能够大幅减少驱动结果集的记录数的过滤条件。
- 嵌套循环连接的另一优点是可以实现连接结果的快速返回响应。这是其他连接方法所不能做到的，因为排序归并连接（sort-merge join）首先要执行排序操作，之后才能进行归并操作以返回数据，而哈希连接（hash join）要等到构建完成驱动结果集所对应的哈希表之后，才能开始返回数据。

举例来说，假设有表 t1 和表 t2，表 t1 有一列为 a，表 t2 有一列为 b，则对于该 SQL 连接语句：

```
SELECT * FROM t1 JOIN t2 ON t1.a = t2.b;
```

如果使用嵌套循环连接算法进行表连接操作，则会按以下伪代码进行连接：

```
For each tuple a in t1 do
    For each tuple b in t2 do
            If t1.a = t2.b
            Then output the tuple <a, b>
```

即对于 t1 表元组的扫描为第一层循环，对于 t2 表元组的扫描为第二层循环，从头开始扫描 t1 表的元组，并在扫描到 t2 表的每个元组时，判断 t1.a 是否等于 t2.b。这样依次扫描下来，就能返回符合条件的元组。

关于嵌套循环连接算法的更详细介绍，参见教材"Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom. Database Systems: The Complete Book. Pearson; 2nd edition (June 5, 2008)"的 15.3 节"Nested-Loop Joins"。

9.3.2 浏览嵌套循环算法源代码

实际上，在 openGauss 数据库中嵌套循环算法的实现源代码并不是如同上述描述算法原理的双重循环那样简单，而是需要处理各种情况。在理解了嵌套循环算法原理的基础上，浏览实现该算法的源代码是提升系统工程能力的一种好方法。

在 openGauss 中，嵌套循环算法源代码位于文件 src/gausskernel/runtime/executor/nodeNestloop.cpp 中的 ExecNestLoop 函数。该函数参数为结构体 NestLoopState 指针，返回值为结构体 TupleTableSlot 指针。这两个结构体的信息将在下一节介绍。ExecNestLoop 函数主要代码流程如图 9.1 所示。

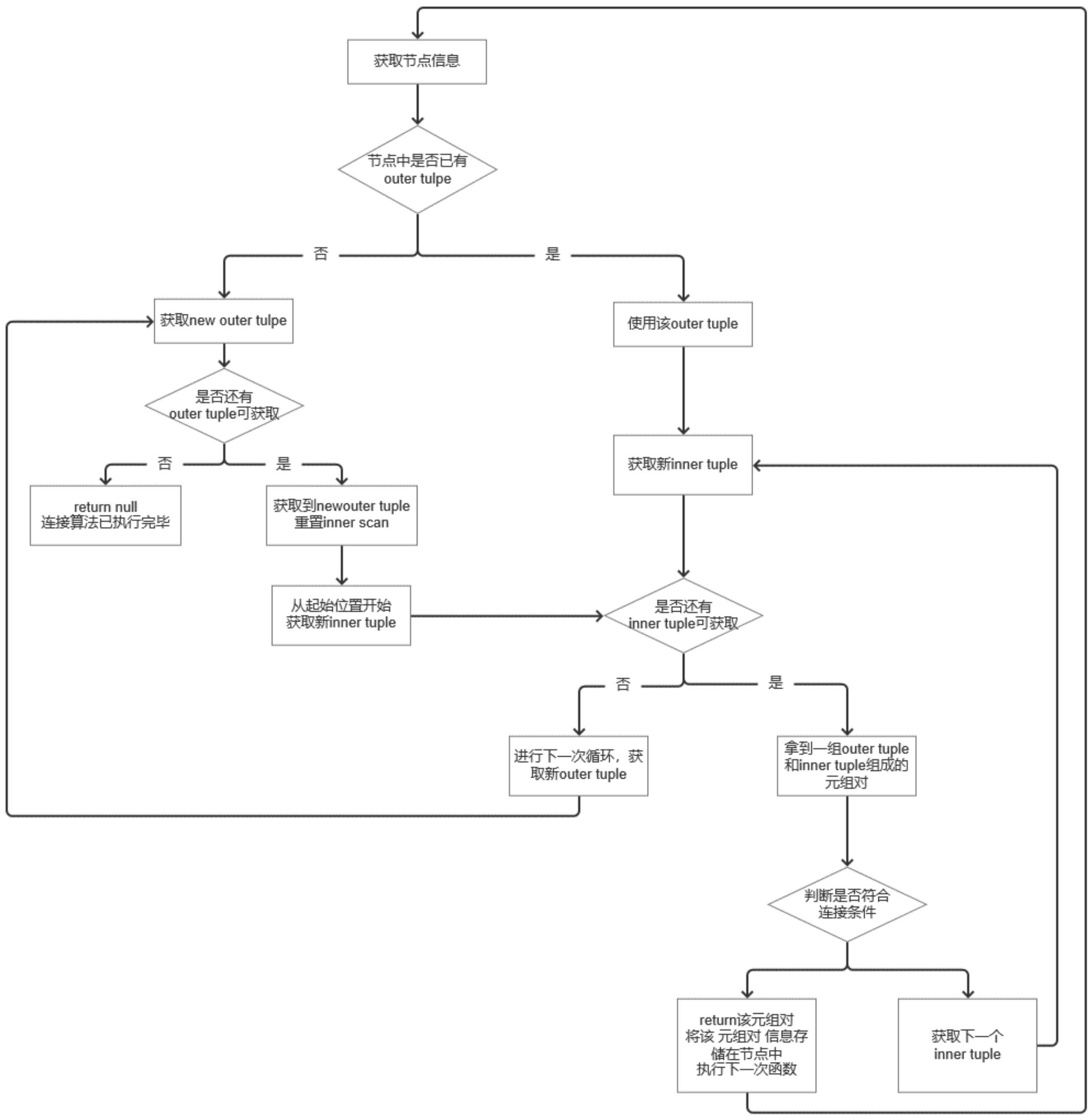

获取节点信息
节点中是否已有 outer tulpe
否
是
获取new outer tulpe
使用该outer tuple
是否还有 outer tuple可获取
否
是
return null 连接算法已执行完毕
获取到newouter tuple 重置inner scan
获取新inner tuple
从起始位置开始 获取新inner tuple
是否还有 inner tuple可获取
否
是
进行下一次循环，获取新outer tuple
拿到一组outer tuple 和inner tuple组成的 元组对
判断是否符合 连接条件
return该元组对 将该 元组对 信息存 储在节点中 执行下一次函数
获取下一个 inner tuple

图 9.1 openGauss 中的嵌套循环算法源代码的主要流程图

9.3.3 相关结构体与源代码

本实验涉及到的几个重要的结构体如下：

1. 结构体 TupleTableSlot

   openGauss 存储层定义的元组数据结构（HeapTuple）和操作接口是面向物理元组的，结构构造和解析开销较大；而执行器（executor）在查询执行过程中通常需要进行属性投影和属性选择过滤判断操作，需要快速获取元组数据。因而，物理元组 HeapTuple 不能满足执行器元组高效处理要求。为此，定义了结构体 MinimalTuple，该结构体去掉了 HeapTuple 中的事务相关字段，因而节省了元组的存储空间占用。TupleTableSlot 是 openGauss 执行器专用的元组存储数据结构。其目的是为了能统一地表示各种形式的元组。TupleTableSlot 本身带有 tts_values、tts_isnull 等字段，用于保存数据。

【源码】src/include/executor/tuptable.h：

```
/* ----------
 * The executor stores tuples in a "tuple table" which is a List of
 * independent TupleTableSlots.  There are several cases we need to handle:
 *      1. physical tuple in a disk buffer page
 *      2. physical tuple constructed in palloc'ed memory
 *      3. "minimal" physical tuple constructed in palloc'ed memory
 *      4. "virtual" tuple consisting of Datum/isnull arrays
 *
 * The first two cases are similar in that they both deal with "materialized"
 * tuples, but resource management is different.  For a tuple in a disk page
 * we need to hold a pin on the buffer until the TupleTableSlot's reference
 * to the tuple is dropped; while for a palloc'd tuple we usually want the
 * tuple pfree'd when the TupleTableSlot's reference is dropped.
 *
 * A "minimal" tuple is handled similarly to a palloc'd regular tuple.
 * At present, minimal tuples never are stored in buffers, so there is no
 * parallel to case 1.  Note that a minimal tuple has no "system columns".
 * (Actually, it could have an OID, but we have no need to access the OID.)
 *
 * A "virtual" tuple is an optimization used to minimize physical data
 * copying in a nest of plan nodes.  Any pass-by-reference Datums in the
 * tuple point to storage that is not directly associated with the
 * TupleTableSlot; generally they will point to part of a tuple stored in
 * a lower plan node's output TupleTableSlot, or to a function result
 * constructed in a plan node's per-tuple econtext.  It is the responsibility
 * of the generating plan node to be sure these resources are not released
 * for as long as the virtual tuple needs to be valid.  We only use virtual
 * tuples in the result slots of plan nodes --- tuples to be copied anywhere
 * else need to be "materialized" into physical tuples.  Note also that a
 * virtual tuple does not have any "system columns".
 *
 * It is also possible for a TupleTableSlot to hold both physical and minimal
 * copies of a tuple.  This is done when the slot is requested to provide
 * the format other than the one it currently holds.  (Originally we attempted
 * to handle such requests by replacing one format with the other, but that
 * had the fatal defect of invalidating any pass-by-reference Datums pointing
 * into the existing slot contents.)  Both copies must contain identical data
 * payloads when this is the case.
 *
 * The Datum/isnull arrays of a TupleTableSlot serve double duty.  When the
 * slot contains a virtual tuple, they are the authoritative data.  When the
 * slot contains a physical tuple, the arrays contain data extracted from
 * the tuple.  (In this state, any pass-by-reference Datums point into
 * the physical tuple.)  The extracted information is built "lazily",
* ie, only as needed.  This serves to avoid repeated extraction of data
* from the physical tuple.
*
* A TupleTableSlot can also be "empty", holding no valid data.  This is
* the only valid state for a freshly-created slot that has not yet had a
* tuple descriptor assigned to it.  In this state, tts_isempty must be
* TRUE, tts_shouldFree FALSE, tts_tuple NULL, tts_buffer InvalidBuffer,
* and tts_nvalid zero.
*
* The tupleDescriptor is simply referenced, not copied, by the TupleTableSlot
* code.  The caller of ExecSetSlotDescriptor() is responsible for providing
* a descriptor that will live as long as the slot does.  (Typically, both
* slots and descriptors are in per-query memory and are freed by memory
* context deallocation at query end; so it's not worth providing any extra
* mechanism to do more.  However, the slot will increment the tupdesc
* reference count if a reference-counted tupdesc is supplied.)
*
* When tts_shouldFree is true, the physical tuple is "owned" by the slot
* and should be freed when the slot's reference to the tuple is dropped.
*
* If tts_buffer is not InvalidBuffer, then the slot is holding a pin
* on the indicated buffer page; drop the pin when we release the
* slot's reference to that buffer.  (tts_shouldFree should always be
* false in such a case, since presumably tts_tuple is pointing at the
* buffer page.)
*
* tts_nvalid indicates the number of valid columns in the tts_values/isnull
* arrays.  When the slot is holding a "virtual" tuple this must be equal
* to the descriptor's natts.  When the slot is holding a physical tuple
* this is equal to the number of columns we have extracted (we always
* extract columns from left to right, so there are no holes).
*
* tts_values/tts_isnull are allocated when a descriptor is assigned to the
* slot; they are of length equal to the descriptor's natts.
*
* tts_mintuple must always be NULL if the slot does not hold a "minimal"
* tuple.  When it does, tts_mintuple points to the actual MinimalTupleData
* object (the thing to be pfree'd if tts_shouldFreeMin is true).  If the slot
* has only a minimal and not also a regular physical tuple, then tts_tuple
* points at tts_minhdr and the fields of that struct are set correctly
* for access to the minimal tuple; in particular, tts_minhdr.t_data points
* MINIMAL_TUPLE_OFFSET bytes before tts_mintuple.  This allows column
* extraction to treat the case identically to regular physical tuples.
*
```

```
 * tts_slow/tts_off are saved state for slot_deform_tuple, and should not
 * be touched by any other code.
 * ----------
 */
typedef struct TupleTableSlot {
    NodeTag type;
    bool tts_isempty;       /* true = slot is empty */
    bool tts_shouldFree;    /* should pfree tts_tuple? */
    bool tts_shouldFreeMin; /* should pfree tts_mintuple? */
    bool tts_slow;          /* saved state for slot_deform_tuple */

    Tuple tts_tuple;    /* physical tuple, or NULL if virtual */
#ifdef PGXC
    /*
     * PGXC extension to support tuples sent from remote Datanode.
     */
    char* tts_dataRow;                   /* Tuple data in DataRow format */
    int tts_dataLen;                     /* Actual length of the data row */
    bool tts_shouldFreeRow;              /* should pfree tts_dataRow? */
   struct AttInMetadata* tts_attinmeta; /* store here info to extract values from the DataRow */
    Oid tts_xcnodeoid;                   /* Oid of node from where the datarow is fetched */
    MemoryContext tts_per_tuple_mcxt;
#endif
    TupleDesc tts_tupleDescriptor; /* slot's tuple descriptor */
    MemoryContext tts_mcxt;        /* slot itself is in this context */
    Buffer tts_buffer;             /* tuple's buffer, or InvalidBuffer */
    int tts_nvalid;                /* # of valid values in tts_values */
    Datum* tts_values;             /* current per-attribute values */
    bool* tts_isnull;              /* current per-attribute isnull flags */
    Datum* tts_lobPointers;
    MinimalTuple tts_mintuple;     /* minimal tuple, or NULL if none */
    HeapTupleData tts_minhdr;      /* workspace for minimal-tuple-only case */
    long tts_off;                  /* saved state for slot_deform_tuple */
    long tts_meta_off;             /* saved state for slot_deform_cmpr_tuple */
    TableAmType tts_tupslotTableAm;    /* slots's tuple table type */
} TupleTableSlot;
```

2. 结构体 NestLoopState

   结构体 NestLoopState 保存了用于进行嵌套循环连接操作所使用的信息，其作为 ExecNestLoop 函数的参数传递。

   结构体 NestLoopState 字段说明：

- nl_NeedNewOuter：布尔值。如果需要新的外层元组，则为 true。
- nl_MatchedOuter：布尔值。如果对于当前外层元组找到了连接匹配，则为 true。
- nl_NullInnerTupleSlot：TupleTableSlot 指针。为左外连接准备的空元组。

【源码】src/include/nodes/execnodes.h

```
/* ----------------
 *   NestLoopState information
 *
 *      NeedNewOuter       true if need new outer tuple on next call
 *      MatchedOuter       true if found a join match for current outer tuple
 *      NullInnerTupleSlot prepared null tuple for left outer joins
 * ----------------
 */
typedef struct NestLoopState {
    JoinState js; /* its first field is NodeTag */
    bool nl_NeedNewOuter;
    bool nl_MatchedOuter;
    bool nl_MaterialAll;
    TupleTableSlot* nl_NullInnerTupleSlot;
} NestLoopState;
```

3. 链表数据结构相关结构体：List 和 ListCell

   openGauss 代码继承了 PostgreSQL 代码中的链表数据结构，涉及到多个元素的存储和访问需求，基本都使用 List 结构体。

- ListCell：存储链表中的一个元素以及指向一个 ListCell 的指针。其中，如果这是一个由 int 或者 Oid 构成的 List，那么 ListCell 直接存储 int 或者 Oid。若不是，则使用 void*来存储，这样可以存储的类型就多了。一般用的时候直接使用强制转换为(Type *)即可使用。next 存储的是下一个 ListCell，由此可以说明 List 是一个线性链表，只能向后寻找。
- List：由 ListCell 组成，List 没有将整个链存储起来，仅存储由 ListCell 组成的线性链表的头和尾。在做查询的时候，也仅仅是通过头进行向后查询。同时还存储了链表的两个属性：链表类型（T_List、T_IntList 或 T_OidList）以及元素（ListCell）的个数。

【源码】src/include/nodes/pg_list.h：

```
typedef struct ListCell ListCell;

typedef struct List {
    NodeTag type; /* T_List, T_IntList, or T_OidList */
    int length;
    ListCell* head;
    ListCell* tail;
} List;

struct ListCell {
    union {
        void* ptr_value;
        int int_value;
        Oid oid_value;
    } data;
    ListCell* next;
};
```

- 宏 foreach(cell, l)和 foreach_cell(cell, l)：用于遍历链表。

```
/*
 * foreach -
 *     a convenience macro which loops through the list
 */
#define foreach(cell, l) for ((cell) = list_head(l); (cell) != NULL; (cell) = lnext(cell))


#define foreach_cell(cell, l) for (ListCell* cell = list_head(l); cell != NULL; cell = lnext(cell))
```

- 宏 lfirst(lc)：获得链表当前元素的值。

```
#define lfirst(lc) ((lc)->data.ptr_value)
```

4. 数据类型 Datum

   Datum 是 openGauss 中函数大量使用的数据类型，主要用于参数传递，它可以表示任何有效的 SQL 类型的数值。其实，Datum 就像 C 语言中的空指针 void *，是一种抽象数据类型。

【源码】src/include/postgres.h：

```
/*
 * Port Notes:
 *  openGauss makes the following assumptions about datatype sizes:
 *
 *  sizeof(Datum) == sizeof(void *) == 4 or 8
 *  sizeof(char) == 1
 *  sizeof(short) == 2
 *
 * When a type narrower than Datum is stored in a Datum, we place it in the
 * low-order bits and are careful that the DatumGetXXX macro for it discards
 * the unused high-order bits (as opposed to, say, assuming they are zero).
 * This is needed to support old-style user-defined functions, since depending
 * on architecture and compiler, the return value of a function returning char
 * or short may contain garbage when called as if it returned Datum.
 */

typedef uintptr_t Datum;

#define SIZEOF_DATUM SIZEOF_VOID_P

typedef Datum* DatumPtr;

#define GET_1_BYTE(datum) (((Datum)(datum)) & 0x000000ff)
#define GET_2_BYTES(datum) (((Datum)(datum)) & 0x0000ffff)
#define GET_4_BYTES(datum) (((Datum)(datum)) & 0xffffffff)
#if SIZEOF_DATUM == 8
#define GET_8_BYTES(datum) ((Datum)(datum))
```

```
#endif
#define SET_1_BYTE(value) (((Datum)(value)) & 0x000000ff)
#define SET_2_BYTES(value) (((Datum)(value)) & 0x0000ffff)
#define SET_4_BYTES(value) (((Datum)(value)) & 0xffffffff)
#if SIZEOF_DATUM == 8
#define SET_8_BYTES(value) ((Datum)(value))
#endif
```

5. 结构体 PlanState

   结构体 PlanState 存储了当前查询的执行计划的状态信息。PlanState 结构体是动态生成的，并且在查询执行过程中不断更新，以根据执行计划的进展来更新查询状态。它通常存储查询执行过程中所需的所有数据。在查询计划树的执行过程中，执行器将使用 PlanState 来记录计划节点的执行状态和数据。每种计划节点（Plan 结构体）都定义了相应的状态节点（PlanState 结构体），所有的状态节点均继承于 PlanState 节点。通过左右子状态节点指针（PlanState 的 lefttree 和 righttree 字段），形成了一棵与查询计划树对应的计划状态树。

   结构体 PlanState 的重要字段说明：

- plan：对应的计划节点指针（Plan*）。
- state：执行器全局状态结构指针（Estate*）。
- targetlist：投影运算相关信息。
- qual：选择运算相关条件。
- lefttree：左子状态节点指针。
- righttree：右子状态节点指针。

【源码】src/include/nodes/execnodes.h：

```
/* ----------------
 *      PlanState node
 *
 * We never actually instantiate any PlanState nodes; this is just the common
 * abstract superclass for all PlanState-type nodes.
 * ----------------
 */
typedef struct PlanState {
    NodeTag type;

    Plan* plan; /* associated Plan node */

    EState* state; /* at execution time, states of individual
                    * nodes point to one EState for the whole
                    * top-level plan */

    Instrumentation* instrument; /* Optional runtime stats for this node */

    /*
     * Common structural data for all Plan types.  These links to subsidiary
     * state trees parallel links in the associated plan tree (except for the
     * subPlan list, which does not exist in the plan tree).
    */
    List* targetlist;           /* target list to be computed at this node */
    List* qual;                 /* implicitly-ANDed qual conditions */
    struct PlanState* lefttree; /* input plan tree(s) */
    struct PlanState* righttree;
    List* initPlan; /* Init SubPlanState nodes (un-correlated expr subselects) */
    List* subPlan;  /* SubPlanState nodes in my expressions */

    /*
     * State for management of parameter-change-driven rescanning
     */
    Bitmapset* chgParam; /* set of IDs of changed Params */
    HbktScanSlot hbktScanSlot;

    /*
     * Other run-time state needed by most if not all node types.
     */
    TupleTableSlot* ps_ResultTupleSlot; /* slot for my result tuples */
    ExprContext* ps_ExprContext;        /* node's expression-evaluation context */
    ProjectionInfo* ps_ProjInfo;        /* info for doing tuple projection */
   bool ps_TupFromTlist;            /* state flag for processing set-valued functions in targetlist
*/

    bool vectorized;  // is vectorized?

    MemoryContext nodeContext; /* Memory Context for this Node */

    bool earlyFreed;                 /* node memory already freed? */
    uint8  stubType;                 /* node stub execution type, see @PlanStubType */
   vectarget_func jitted_vectarget; /* LLVM IR function pointer to point to the codegened targetlist
expr. */

    /*
     * Describe issues found in curernt plan node, mainly used for issue de-duplication
     * of data skew and inaccurate e-rows
     */
    List* plan_issues;
    bool recursive_reset; /* node already reset? */
    bool qual_is_inited;

    bool do_not_reset_rownum;
    int64 ps_rownum;    /* store current rownum */
} PlanState;
```

6. 结构体 ExprContext

   结构体 ExprContext 是保存表达式内存上下文的数据结构，表达式计算过程中的参数以及表达式所使用的内存上下文都会存放到此结构中。

【源码】src/include/nodes/execnodes.h：

```
/* ----------------
 *    ExprContext
 *
 *      This class holds the "current context" information
 *      needed to evaluate expressions for doing tuple qualifications
 *      and tuple projections.  For example, if an expression refers
 *      to an attribute in the current inner tuple then we need to know
 *      what the current inner tuple is and so we look at the expression
 *      context.
 *
 *  There are two memory contexts associated with an ExprContext:
 *  * ecxt_per_query_memory is a query-lifespan context, typically the same
 *    context the ExprContext node itself is allocated in.  This context
 *    can be used for purposes such as storing function call cache info.
 *  * ecxt_per_tuple_memory is a short-term context for expression results.
 *    As the name suggests, it will typically be reset once per tuple,
 *    before we begin to evaluate expressions for that tuple.  Each
 *    ExprContext normally has its very own per-tuple memory context.
 *
 *  CurrentMemoryContext should be set to ecxt_per_tuple_memory before
 *  calling ExecEvalExpr() --- see ExecEvalExprSwitchContext().
 * ----------------
 */
struct PLpgSQL_execstate;

typedef struct ExprContext {
    NodeTag type;

    /* Tuples that Var nodes in expression may refer to */
    TupleTableSlot* ecxt_scantuple;
    TupleTableSlot* ecxt_innertuple;
    TupleTableSlot* ecxt_outertuple;

    /* Memory contexts for expression evaluation --- see notes above */
    MemoryContext ecxt_per_query_memory;
    MemoryContext ecxt_per_tuple_memory;

    /* Values to substitute for Param nodes in expression */
    ParamExecData* ecxt_param_exec_vals; /* for PARAM_EXEC params */
    ParamListInfo ecxt_param_list_info;  /* for other param types */
```

```
/*
 * Values to substitute for Aggref nodes in the expressions of an Agg
 * node, or for WindowFunc nodes within a WindowAgg node.
 */
Datum* ecxt_aggvalues; /* precomputed values for aggs/windowfuncs */
bool* ecxt_aggnulls;   /* null flags for aggs/windowfuncs */

/* Value to substitute for CaseTestExpr nodes in expression */
Datum caseValue_datum;
bool caseValue_isNull;

ScalarVector* caseValue_vector;

/* Value to substitute for CoerceToDomainValue nodes in expression */
Datum domainValue_datum;
bool domainValue_isNull;

/* Link to containing EState (NULL if a standalone ExprContext) */
struct EState* ecxt_estate;

/* Functions to call back when ExprContext is shut down */
ExprContext_CB* ecxt_callbacks;

// vector specific fields
// consider share space with row fields
//
VectorBatch* ecxt_scanbatch;
VectorBatch* ecxt_innerbatch;
VectorBatch* ecxt_outerbatch;

// Batch to substitute for Aggref nodes in the expression of an VecAgg node
//
VectorBatch* ecxt_aggbatch;

/*
 * mark the real rows for expression cluster, all the results' m_rows generated by
 * vec-expression are aligned by econtext->align_rows
 */
int align_rows;

bool m_fUseSelection;  // Shall we use selection vector?
ScalarVector* qual_results;
ScalarVector* boolVector;
```

```
    bool is_cursor;
  Cursor_Data cursor_data;
    int dno;
    PLpgSQL_execstate* plpgsql_estate;
    /*
     * For vector set-result function.
     */
    bool have_vec_set_fun;
    bool* vec_fun_sel;  // selection for vector set-result function.
    int current_row;
} ExprContext;
```

7. 函数 ExecInitNestLoop
   一条 SQL 语句在执行过程中会调用 PortalStart 函数遍历整个查询计划树(plan tree)，对每个算子进行初始化。算子初始化函数的命名规则为"ExecInit+算子名"的形式。因而，嵌套循环算子的初始化函数名为 ExecInitNestloop。初始化函数中首先会根据对应的 Plan 结构初始化一个对应的 PlanState 结构，这个结构是执行过程中的核心数据结构，包含了在执行过程中需要用的一些数据存储空间以及执行信息。

【源码】src/gausskernel/runtime/executor/nodeNestloop.cpp：

```
/* ----------------------------------------------------------------
 *      ExecInitNestLoop
 * ----------------------------------------------------------------
 */
NestLoopState* ExecInitNestLoop(NestLoop* node, EState* estate, int eflags)
{
    /* check for unsupported flags */
    Assert(!(eflags & (EXEC_FLAG_BACKWARD | EXEC_FLAG_MARK)));

    NL1_printf("ExecInitNestLoop: %s\n", "initializing node");

    /*
     * create state structure
     */
    NestLoopState* nlstate = makeNode(NestLoopState);
    nlstate->js.ps.plan = (Plan*)node;
    nlstate->js.ps.state = estate;
    nlstate->nl_MaterialAll = node->materialAll;

    /*
     * Miscellaneous initialization
     *
     * create expression context for node
     */
    ExecAssignExprContext(estate, &nlstate->js.ps);
    /*
   * initialize child expressions
     */
    nlstate->js.ps.targetlist = (List*)ExecInitExpr((Expr*)node->join.plan.targetlist,
(PlanState*)nlstate);
   nlstate->js.ps.qual = (List*)ExecInitExpr((Expr*)node->join.plan.qual, (PlanState*)nlstate);
    nlstate->js.jointype = node->join.jointype;
   nlstate->js.joinqual = (List*)ExecInitExpr((Expr*)node->join.joinqual, (PlanState*)nlstate);
    Assert(node->join.nulleqqual == NIL);

    /*
     * initialize child nodes
     *
     * If we have no parameters to pass into the inner rel from the outer,
     * tell the inner child that cheap rescans would be good.  If we do have
     * such parameters, then there is no point in REWIND support at all in the
     * inner child, because it will always be rescanned with fresh parameter
     * values.
     */
    outerPlanState(nlstate) = ExecInitNode(outerPlan(node), estate, eflags);
    if (node->nestParams == NIL)
        eflags |= EXEC_FLAG_REWIND;
    else
        eflags &= ~EXEC_FLAG_REWIND;
    innerPlanState(nlstate) = ExecInitNode(innerPlan(node), estate, eflags);

    /*
     * tuple table initialization
     */
    ExecInitResultTupleSlot(estate, &nlstate->js.ps);

    switch (node->join.jointype) {
        case JOIN_INNER:
        case JOIN_SEMI:
            break;
        case JOIN_LEFT:
        case JOIN_ANTI:
        case JOIN_LEFT_ANTI_FULL:
            nlstate->nl_NullInnerTupleSlot = ExecInitNullTupleSlot(estate,
ExecGetResultType(innerPlanState(nlstate)));
            break;
        default:
            ereport(ERROR,
                (errcode(ERRCODE_UNRECOGNIZED_NODE_TYPE),
                    errmodule(MOD_EXECUTOR),
                    errmsg("unrecognized join type: %d when initializing nestLoop",
(int)node->join.jointype)));
    }

    /*
     * initialize tuple type and projection info
     * the result in this case would hold only virtual data.
     */
    ExecAssignResultTypeFromTL(&nlstate->js.ps, TAM_HEAP);

    ExecAssignProjectionInfo(&nlstate->js.ps, NULL);

    /*
     * finally, wipe the current outer tuple clean.
     */
    nlstate->js.ps.ps_TupFromTlist = false;
    nlstate->nl_NeedNewOuter = true;
    nlstate->nl_MatchedOuter = false;

    NL1_printf("ExecInitNestLoop: %s\n", "node initialized");

    return nlstate;
}
```

8. 函数 ExecNestLoop
   一条 SQL 语句在执行过程中会调用 PortalRun 函数进行查询执行计划的运行。在执行过程中，所有执行算子分为 2 大类，行存储算子和向量化算子。这两类算子分别对应行存储执行引擎和向量化执行引擎。我们只关注行存储执行引擎。行存储算子执行入口函数的命名规则是“Exec+算子名”的形式。因而，嵌套循环算子的执行函数名为 ExecNestLoop。该函数的具体代码见 9.4.1 节。
9. 函数 ExecEndNestLoop
   一条 SQL 语句在执行过程中会调用 PortalDrop 函数对各个算子进行递归清理，主要是清理在执行过程中产生的内存。各个算子的清理函数入口命名规则是“ExecEnd+算子名”。因而，嵌套循环算子清理函数名为 ExecEndNestLoop。

【源码】src/gausskernel/runtime/executor/nodeNestloop.cpp：

```
/* ----------------------------------------------------------------
 *      ExecEndNestLoop
 *
 *      closes down scans and frees allocated storage
 * ----------------------------------------------------------------
 */
void ExecEndNestLoop(NestLoopState* node)
{
    NL1_printf("ExecEndNestLoop: %s\n", "ending node processing");
    /*
     * Free the exprcontext
     */
    ExecFreeExprContext(&node->js.ps);

    /*
     * clean out the tuple table
     */
    (void)ExecClearTuple(node->js.ps.ps_ResultTupleSlot);

    /*
     * close down subplans
     */
    ExecEndNode(outerPlanState(node));
    ExecEndNode(innerPlanState(node));

    NL1_printf("ExecEndNestLoop: %s\n", "node processing ended");
}
```

9.4 实验步骤

9.4.1 嵌套循环算法源代码定位

1. 确定嵌套循环算法源代码的位置。
   首先，要在 openGauss 中定位到实现嵌套循环算法源代码所在文件和函数。经查找，该部分代码位于:

文件：src/gausskernel/runtime/executor/nodeNestloop.cpp
函数：ExecNestLoop

```
/* ----------------------------------------------------------------
 *       ExecNestLoop(node)
 *
 * old comments
 *       Returns the tuple joined from inner and outer tuples which
 *       satisfies the qualification clause.
 *
 *       It scans the inner relation to join with current outer tuple.
 *
 *       If none is found, next tuple from the outer relation is retrieved
 *       and the inner relation is scanned from the beginning again to join
 *       with the outer tuple.
 *
 *       NULL is returned if all the remaining outer tuples are tried and
 *       all fail to join with the inner tuples.
 *
 *       NULL is also returned if there is no tuple from inner relation.
 *
 *       Conditions:
 *         -- outerTuple contains current tuple from outer relation and
 *             the right son(inner relation) maintains "cursor" at the tuple
 *             returned previously.
 *                 This is achieved by maintaining a scan position on the outer
 *                 relation.
 *
 *       Initial States:
 *         -- the outer child and the inner child
 *               are prepared to return the first tuple.
 * ----------------------------------------------------------------
 */
TupleTableSlot* ExecNestLoop(NestLoopState* node)
{
    TupleTableSlot* outer_tuple_slot = NULL;
    TupleTableSlot* inner_tuple_slot = NULL;
    ListCell* lc = NULL;

    /*
     * get information from the node
     */
    ENL1_printf("getting info from node");

    NestLoop* nl = (NestLoop*)node->js.ps.plan;
    List* joinqual = node->js.joinqual;
    List* otherqual = node->js.ps.qual;
    PlanState* outer_plan = outerPlanState(node);
    PlanState* inner_plan = innerPlanState(node);
    ExprContext* econtext = node->js.ps.ps_ExprContext;

    /*
     * Check to see if we're still projecting out tuples from a previous join
     * tuple (because there is a function-returning-set in the projection
     * expressions).  If so, try to project another one.
     */
    if (node->js.ps.ps_TupFromTlist) {
        ExprDoneCond is_done;

        TupleTableSlot* result = ExecProject(node->js.ps.ps_ProjInfo, &is_done);
        if (is_done == ExprMultipleResult)
```

```
            return result;
      /* Done with that source tuple... */
        node->js.ps.ps_TupFromTlist = false;
    }

    /*
     * Reset per-tuple memory context to free any expression evaluation
     * storage allocated in the previous tuple cycle.  Note this can't happen
     * until we're done projecting out tuples from a join tuple.
     */
    ResetExprContext(econtext);

    /*
     * Ok, everything is setup for the join so now loop until we return a
     * qualifying join tuple.
     */
    ENL1_printf("entering main loop");

    if (node->nl_MaterialAll) {
        MaterialAll(inner_plan);
        node->nl_MaterialAll = false;
    }

    for (;;) {
        /*
         * If we don't have an outer tuple, get the next one and reset the
         * inner scan.
         */
        if (node->nl_NeedNewOuter) {
            ENL1_printf("getting new outer tuple");
            outer_tuple_slot = ExecProcNode(outer_plan);
            /*
             * if there are no more outer tuples, then the join is complete..
             */
            if (TupIsNull(outer_tuple_slot)) {
                ExecEarlyFree(inner_plan);
                ExecEarlyFree(outer_plan);

                EARLY_FREE_LOG(elog(LOG,
                    "Early Free: NestLoop is done "
                    "at node %d, memory used %d MB.",
                    (node->js.ps.plan)->plan_node_id,
                    getSessionMemoryUsageMB()));
                ENL1_printf("no outer tuple, ending join");

                return NULL;
            }

            ENL1_printf("saving new outer tuple information");
            econtext->ecxt_outertuple = outer_tuple_slot;
            node->nl_NeedNewOuter = false;
            node->nl_MatchedOuter = false;

            /*
             * fetch the values of any outer Vars that must be passed to the
             * inner scan, and store them in the appropriate PARAM_EXEC slots.
             */
            foreach (lc, nl->nestParams) {
                NestLoopParam* nlp = (NestLoopParam*)lfirst(lc);
                int paramno = nlp->paramno;
                ParamExecData* prm = NULL;

                prm = &(econtext->ecxt_param_exec_vals[paramno]);
                /* Param value should be an OUTER_VAR var */
                Assert(IsA(nlp->paramval, Var));
                Assert(nlp->paramval->varno == OUTER_VAR);
                Assert(nlp->paramval->varattno > 0);
                Assert(outer_tuple_slot != NULL && outer_tuple_slot->tts_tupleDescriptor !=
NULL);
                /* Get the Table Accessor Method*/
                prm->value = tableam_tslot_getattr(outer_tuple_slot, nlp->paramval->varattno,
&(prm->isnull));
                /*
                 * the following two parameters are called when there exist
                 * join-operation with column table (see ExecEvalVecParamExec).
                 */
                prm->valueType = outer_tuple_slot->tts_tupleDescriptor->tdtypeid;
                prm->isChanged = true;
                /* Flag parameter value as changed */
                inner_plan->chgParam = bms_add_member(inner_plan->chgParam, paramno);
            }

            /*
             * now rescan the inner plan
             */
            ENL1_printf("rescanning inner plan");
            ExecReScan(inner_plan);
        }

        /*
         * we have an outerTuple, try to get the next inner tuple.
         */
        ENL1_printf("getting new inner tuple");

        /*
         * If inner plan is mergejoin, which does not cache data,
         * but will early free the left and right tree's caching memory.
         * When rescan left tree, may fail.
         */
        bool orig_value = inner_plan->state->es_skip_early_free;
        if (!IsA(inner_plan, MaterialState))
            inner_plan->state->es_skip_early_free = true;

        inner_tuple_slot = ExecProcNode(inner_plan);

        inner_plan->state->es_skip_early_free = orig_value;
        econtext->ecxt_innertuple = inner_tuple_slot;

        if (TupIsNull(inner_tuple_slot)) {
            ENL1_printf("no inner tuple, need new outer tuple");

            node->nl_NeedNewOuter = true;

            if (!node->nl_MatchedOuter && (node->js.jointype == JOIN_LEFT || node->js.jointype ==
JOIN_ANTI ||
                                              node->js.jointype == JOIN_LEFT_ANTI_FULL)) {
                /*
                 * We are doing an outer join and there were no join matches
                 * for this outer tuple.  Generate a fake join tuple with
                 * nulls for the inner tuple, and return it if it passes the
                 * non-join quals.
                 */
                econtext->ecxt_innertuple = node->nl_NullInnerTupleSlot;

                ENL1_printf("testing qualification for outer-join tuple");

                if (otherqual == NIL || ExecQual(otherqual, econtext, false)) {
                    /*
                     * qualification was satisfied so we project and return
                     * the slot containing the result tuple using
                     * function ExecProject.
                     */
                    ExprDoneCond is_done;

                    ENL1_printf("qualification succeeded, projecting tuple");

                    TupleTableSlot* result = ExecProject(node->js.ps.ps_ProjInfo, &is_done);

                    if (is_done != ExprEndResult) {
                        node->js.ps.ps_TupFromTlist = (is_done == ExprMultipleResult);
                        return result;
                    }
                } else
                    InstrCountFiltered2(node, 1);
            }

            /*
             * Otherwise just return to top of loop for a new outer tuple.
             */
            continue;
        }

        /*
         * at this point we have a new pair of inner and outer tuples so we
         * test the inner and outer tuples to see if they satisfy the node's
         * qualification.
         *
         * Only the joinquals determine MatchedOuter status, but all quals
         * must pass to actually return the tuple.
         */
        ENL1_printf("testing qualification");

        if (ExecQual(joinqual, econtext, false)) {
            node->nl_MatchedOuter = true;

            /* In an antijoin, we never return a matched tuple */
            if (node->js.jointype == JOIN_ANTI || node->js.jointype == JOIN_LEFT_ANTI_FULL) {
                node->nl_NeedNewOuter = true;
                continue; /* return to top of loop */
            }

            /*
             * In a semijoin, we'll consider returning the first match, but
             * after that we're done with this outer tuple.
             */
```

```
            if (node->js.jointype == JOIN_SEMI)
                node->nl_NeedNewOuter = true;

            if (otherqual == NIL || ExecQual(otherqual, econtext, false)) {
                /*
                 * qualification was satisfied so we project and return the
                 * slot containing the result tuple using ExecProject().
                 */
                ExprDoneCond is_done;

                ENL1_printf("qualification succeeded, projecting tuple");

                TupleTableSlot* result = ExecProject(node->js.ps.ps_ProjInfo, &is_done);

                if (is_done != ExprEndResult) {
                    node->js.ps.ps_TupFromTlist = (is_done == ExprMultipleResult);
                    /*
                     * @hdfs
                     * Optimize plan by informational constraint.
                     */
                    if (((NestLoop*)(node->js.ps.plan))->join.optimizable) {
                        node->nl_NeedNewOuter = true;
                    }

                    return result;
                }
            } else
                InstrCountFiltered2(node, 1);
        } else
            InstrCountFiltered1(node, 1);

        /*
         * Tuple fails qual, so free per-tuple memory and try again.
         */
        ResetExprContext(econtext);

        ENL1_printf("qualification failed, looping");
    }
}
```

2. 仔细阅读 ExecNestLoop 函数源代码，与实验原理 9.3.2 节中的流程图（图 9.1）进行对照分析。理解 ExecNestLoop 函数源代码是如何实现的嵌套循环连接算法。

9.4.2 连接数据库：准备查询数据

1. 以 dblab 用户登录主机，使用 gsql 连接 railway 数据库。

```
[dblab@eduog openGauss-server-v3.0.0]$ gsql railway -r
gsql ((openGauss 3.0.0 build ) compiled at 2023-01-29 11:22:50 commit 0 last mr  debug)
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

railway=#
```

2. 将之前实验在 railway 数据库中建的所有表（如果有）删除。

```
DROP TABLE orders;
DROP TABLE trainrun;
DROP TABLE trainstop;
DROP TABLE train;
DROP TABLE station;
DROP TABLE users;
DROP TABLE users2;
```

3. 创建表并插入数据。

   创建 users 表：

```
CREATE TABLE users
(
   u_id varchar(20),                -- 用户 id，用于系统登录账户名（主键）
   u_passwd varchar(20),            -- 密码，用于系统登录密码
   u_name varchar(10),              -- 真实姓名
   u_idnum varchar(20),             -- 证件号码
   u_regtime timestamp,             -- 注册时间
   CONSTRAINT pk_users PRIMARY KEY (u_id)
);
```

   创建 orders 表：

```
CREATE TABLE orders
(
  o_id int,                         -- 订单 id（主键）
  o_uid varchar(20),                -- 用户 id（外键：参照 users(u_id)）
  o_tdate date,                     -- 发车日期
  o_tid varchar(10),                -- 车次（外键：参照 train(t_id)）
  o_sstation varchar(20),           -- 上车站（外键：参照 station(s_name)）
  o_estation varchar(20),           -- 下车站（外键：参照 station(s_name)）
  o_seattype smallint,              -- 座位类型：一等 1、二等 2
  o_carriage smallint,              -- 车厢号
  o_seatnum smallint,               -- 座位号（排）
  o_seatloc char(1),                -- 座位位置：ABCEF
  o_price money,                    -- 订单金额
  o_ispaid boolean,                 -- 是否已支付
  o_ctime timestamp,                -- 订单创建时间
  CONSTRAINT pk_orders PRIMARY KEY (o_id)
);
```

添加外键约束：

```
ALTER TABLE orders ADD CONSTRAINT fk_orders_users FOREIGN KEY (o_uid) REFERENCES users(u_id);
```

插入 users 表的数据：

```
INSERT INTO users VALUES('1','qweasd','张三','123456789', '2000-06-23 12:00:00');
INSERT INTO users VALUES('2','qweasd','李四','123456712', '2000-06-24 13:00:00');
INSERT INTO users VALUES('3','qweasd','王五','123456754', '2000-06-25 14:00:00');
```

插入 orders 表的数据：

```
INSERT INTO orders VALUES(1,'1','2022-04-29','C2002','天津','北京南',2,8,7,'F',54,1,'2022-04-27
16:00:12');
INSERT INTO orders VALUES(2,'3','2022-04-29','G321','天津南','福',1,4,7,'A',742.5,1,'2022-04-27
17:00:12');
INSERT INTO orders VALUES(3,'3','2022-04-29','G1709','天津西','重庆',2,9,3,'D',929,1,'2022-04-27
18:00:12');
```

4. 执行测试查询。

查询 users 表数据：

```
railway=# SELECT u_id, u_name FROM users;
 u_id | u_name
------+--------
 1    | 张三
 2    | 李四
 3    | 王五
(3 rows)
```

查询 orders 表数据：

```
railway=# SELECT o_id, o_uid, o_tid FROM orders;
 o_id | o_uid | o_tid
------+-------+-------
    1 | 1     | C2002
    2 | 3     | G321
    3 | 3     | G1709
(3 rows)
```

9.4.3 添加代码：输出 ExecNestLoop 函数调试信息

1. 对嵌套循环算法代码 ExecNestLoop 函数中的相关调试信息进行输出。

观察发现，ExecNestLoop 函数中的若干位置都存在输出调试信息的宏调用，诸如：

```
ENL1_printf("getting info from node");
```

经查找，宏 ENL1_printf 的定义代码如下：

```
/* ----------------
 *        nest loop debugging defines
 * ----------------
 */
#ifdef EXEC_NESTLOOPDEBUG
```

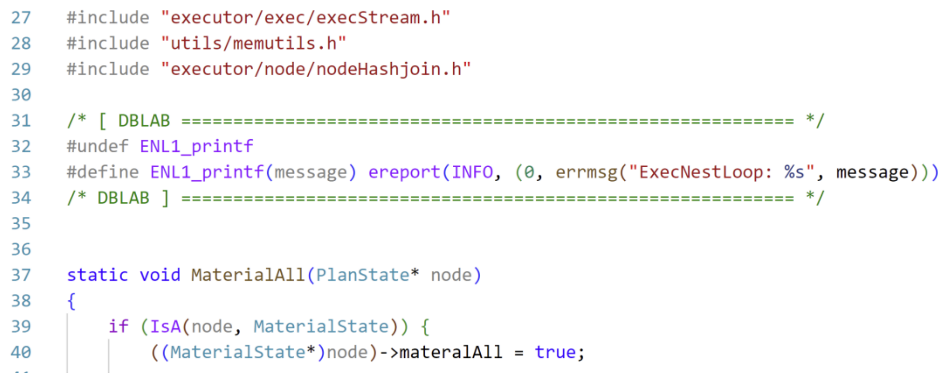
```
#define NL_nodeDisplay(l) nodeDisplay(l)
#define NL_printf(s) printf(s)
#define NL1_printf(s, a) printf(s, a)
#define ENL1_printf(message) printf("ExecNestLoop: %s\n", message)
#else
#define NL_nodeDisplay(l)
#define NL_printf(s)
#define NL1_printf(s, a)
#define ENL1_printf(message)
#endif /* EXEC_NESTLOOPDEBUG */
```

可知, 宏 ENL1_printf 就是专门用来进行 Nestloop 算子调试的, 只是在默认情况下, EXEC_NESTLOOPDEBUG 变量没有被定义, 因而, ENL1_printf(message)被定义为了空白, 即 ExecNestLoop 函数中的 ENL1_printf 宏调用并不会输出信息。

我们在 nodeNestloop.cpp 的开头预处理指令部分添加以下代码, 将 ENL1_printf 宏重定义为用 ereport 函数输出信息。这样就可以在执行 ExecNestLoop 函数时, 将其中的调试信息输出到 gsql 了。

```
/* [ DBLAB ========================================================== */
#undef ENL1_printf
#define ENL1_printf(message) ereport(INFO, (0, errmsg("ExecNestLoop: %s", message)))
/* DBLAB ] ========================================================== */
```

该段代码的添加位置如图 9.2 所示。



```
#include "executor/exec/execStream.h"
#include "utils/memutils.h"
#include "executor/node/nodeHashjoin.h"

/* [ DBLAB ========================================================== */
#undef ENL1_printf
#define ENL1_printf(message) ereport(INFO, (0, errmsg("ExecNestLoop: %s", message)))
/* DBLAB ] ========================================================== */


static void MaterialAll(PlanState* node)
{
    if (IsA(node, MaterialState)) {
        ((MaterialState*)node)->materalAll = true;
```

图 9.2 添加 ENL1_printf 宏定义以输出 ExecNestLoop 函数调试信息

2. 停止 openGauss 服务:

```
[dblab@eduog openGauss-server-v3.0.0]$ gs_ctl stop -D $GAUSSHOME/data -Z single_node -l logfile
```

3. 对 openGauss 进行编译和安装:

```
[dblab@eduog openGauss-server-v3.0.0]$ make -j4
[dblab@eduog openGauss-server-v3.0.0]$ make install
```

4. 启动 openGauss:

```
[dblab@eduog openGauss-server-v3.0.0]$ gs_ctl start -D $GAUSSHOME/data -Z single_node -l logfile
```

5. 使用 gsql 再次连接 railway 数据库, 执行 users 表与 orders 表的连接查询:

```
railway=# SELECT u_id, u_name, o_id, o_uid, o_tid FROM users INNER JOIN orders ON u_id = o_uid;
 u_id | u_name | o_id | o_uid | o_tid
------+--------+------+-------+-------
 1    | 张三   |    1 | 1     | C2002
 3    | 王五   |    2 | 3     | G321
 3    | 王五   |    3 | 3     | G1709
(3 rows)
```
但发现没有输出预期的 ExecNestLoop 中的调试信息！那只有一种可能，即 ExecNestLoop 函数根本没有被执行。查看一条 SQL 语句是具体如何被执行的，即查看具体的查询执行计划，可使用 EXPLAIN 语句。

6. 执行 EXPLAIN 语句，查看上述连接查询的执行计划：

```
railway=# EXPLAIN SELECT u_id, u_name, o_id, o_uid, o_tid FROM users INNER JOIN orders ON u_id =
o_uid;
                             QUERY PLAN
---------------------------------------------------------------------
 Hash Join  (cost=16.30..33.12 rows=287 width=196)
   Hash Cond: ((orders.o_uid)::text = (users.u_id)::text)
   ->  Seq Scan on orders  (cost=0.00..12.87 rows=287 width=100)
   ->  Hash  (cost=12.80..12.80 rows=280 width=96)
         ->  Seq Scan on users  (cost=0.00..12.80 rows=280 width=96)
(5 rows)
```
经观察，果然查询执行计划显示使用的是 Hash 连接（hash join），而不是嵌套循环连接。采用传统的表连接语法，发现 EXPLAIN 的结果是一样的。
（关于 EXPLAIN 语言输出的查询执行计划的解释，可参见文档 https://www.postgresql.org/docs/current/using-explain.html）

```
railway=# SELECT u_id, u_name, o_id, o_uid, o_tid FROM users, orders WHERE u_id = o_uid;
 u_id | u_name | o_id | o_uid | o_tid
------+--------+------+-------+-------
 1    | 张三   |    1 | 1     | C2002
 3    | 王五   |    2 | 3     | G321
 3    | 王五   |    3 | 3     | G1709
(3 rows)

railway=# EXPLAIN SELECT u_id, u_name, o_id, o_uid, o_tid FROM users, orders WHERE u_id = o_uid;
                             QUERY PLAN
---------------------------------------------------------------------
 Hash Join  (cost=16.30..33.12 rows=287 width=196)
   Hash Cond: ((orders.o_uid)::text = (users.u_id)::text)
   ->  Seq Scan on orders  (cost=0.00..12.87 rows=287 width=100)
   ->  Hash  (cost=12.80..12.80 rows=280 width=96)
         ->  Seq Scan on users  (cost=0.00..12.80 rows=280 width=96)
(5 rows)
```
openGauss 选择了使用基于哈希的连接（Hash Join）算法，而非嵌套循环连接（Nested Loop Join）算法。
（关于 Hash Join 连接算法的详细介绍，参见教材“Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom. Database Systems: The Complete Book. Pearson; 2nd edition (June 5, 2008)”15.5 节 Two-Pass Algorithms Based on Hashing）

7. 通过 SET 配置参数 enable_hashjoin 为 off，禁止使用 Hash Join 连接算法。
（关于查询执行计划相关的参数选项，可参见文档

https://www.postgresql.org/docs/current/runtime-config-query.html）

```
railway=# SET enable_hashjoin = off;
SET
```

8. 再次执行查询：

```
railway=# SELECT u_id, u_name, o_id, o_uid, o_tid FROM users INNER JOIN orders ON u_id = o_uid;
 u_id | u_name | o_id | o_uid | o_tid
------+--------+------+-------+-------
 1    | 张三   |    1 | 1     | C2002
 3    | 王五   |    2 | 3     | G321
 3    | 王五   |    3 | 3     | G1709
(3 rows)
```

发现还是没有输出预期的 ExecNestLoop 中的调试信息！

9. 执行 EXPLAIN 语句，查看查询执行计划：

```
railway=# EXPLAIN SELECT u_id, u_name, o_id, o_uid, o_tid FROM users INNER JOIN orders ON u_id =
o_uid;
                              QUERY PLAN
-----------------------------------------------------------------------
 Merge Join  (cost=48.77..54.47 rows=287 width=196)
   Merge Cond: ((users.u_id)::text = (orders.o_uid)::text)
   ->  Sort  (cost=24.18..24.88 rows=280 width=96)
         Sort Key: users.u_id
         ->  Seq Scan on users  (cost=0.00..12.80 rows=280 width=96)
   ->  Sort  (cost=24.59..25.30 rows=287 width=100)
         Sort Key: orders.o_uid
         ->  Seq Scan on orders  (cost=0.00..12.87 rows=287 width=100)
(8 rows)
```

这次，openGauss 选择了使用排序归并连接（Sort-Merge Join）算法，还不是嵌套循环连接（Nested Loop Join）算法。

（关于 Merge Join 连接算法的详细介绍，参见教材"Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom. Database Systems: The Complete Book. Pearson; 2nd edition (June 5, 2008)" 15.4 节 Two-Pass Algorithms Based on Sorting）

10. 通过 SET 配置参数 enable_mergejoin 为 off，禁止使用 Merge Join 连接算法：

```
railway=# SET enable_mergejoin = off;
SET
```

11. 再次执行连接查询：

```
SELECT u_id, u_name, o_id, o_uid, o_tid FROM users INNER JOIN orders ON u_id = o_uid;
```

成功输出了 ExecNestLoop 函数中的调试信息，如下：

```
railway=# SELECT u_id, u_name, o_id, o_uid, o_tid FROM users INNER JOIN orders ON u_id = o_uid;
INFO:  ExecNestLoop: getting info from node
INFO:  ExecNestLoop: entering main loop
INFO:  ExecNestLoop: getting new outer tuple
INFO:  ExecNestLoop: saving new outer tuple information
INFO:  ExecNestLoop: rescanning inner plan
INFO:  ExecNestLoop: getting new inner tuple
```

```
INFO:  ExecNestLoop: testing qualification
INFO:  ExecNestLoop: qualification succeeded, projecting tuple
INFO:  ExecNestLoop: getting info from node
INFO:  ExecNestLoop: entering main loop
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: no inner tuple, need new outer tuple
INFO:  ExecNestLoop: getting new outer tuple
INFO:  ExecNestLoop: saving new outer tuple information
INFO:  ExecNestLoop: rescanning inner plan
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  ExecNestLoop: qualification succeeded, projecting tuple
INFO:  ExecNestLoop: getting info from node
INFO:  ExecNestLoop: entering main loop
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: no inner tuple, need new outer tuple
INFO:  ExecNestLoop: getting new outer tuple
INFO:  ExecNestLoop: saving new outer tuple information
INFO:  ExecNestLoop: rescanning inner plan
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  ExecNestLoop: qualification succeeded, projecting tuple
INFO:  ExecNestLoop: getting info from node
INFO:  ExecNestLoop: entering main loop
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: no inner tuple, need new outer tuple
INFO:  ExecNestLoop: getting new outer tuple
INFO:  ExecNestLoop: no outer tuple, ending join
 u_id | u_name | o_id | o_uid | o_tid
------+--------+------+-------+-------
 1    | 张三   |    1 | 1     | C2002
 3    | 王五   |    2 | 3     | G321
 3    | 王五   |    3 | 3     | G1709
(3 rows)
```

12. 执行 EXPLAIN 语句，查看查询执行计划：

```
railway=# EXPLAIN SELECT u_id, u_name, o_id, o_uid, o_tid FROM users INNER JOIN orders ON u_id =
o_uid;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Nested Loop  (cost=0.00..141.23 rows=287 width=196)
   ->  Seq Scan on orders  (cost=0.00..12.87 rows=287 width=100)
   ->  Index Scan using pk_users on users  (cost=0.00..0.44 rows=1 width=96)
         Index Cond: ((u_id)::text = (orders.o_uid)::text)
(4 rows)
```

发现这次执行 openGauss 终于使用了 Nested Loop 连接算法！通过观察，发现此查询执行计划中，对于 orders 表进行了顺序扫描（Seq Scan）操作，而对于 users 表进行的是索引扫描（Index Scan）操作。使用了名为 pk_users 索引，该索引正是 users 表的主键对应的默认索引。

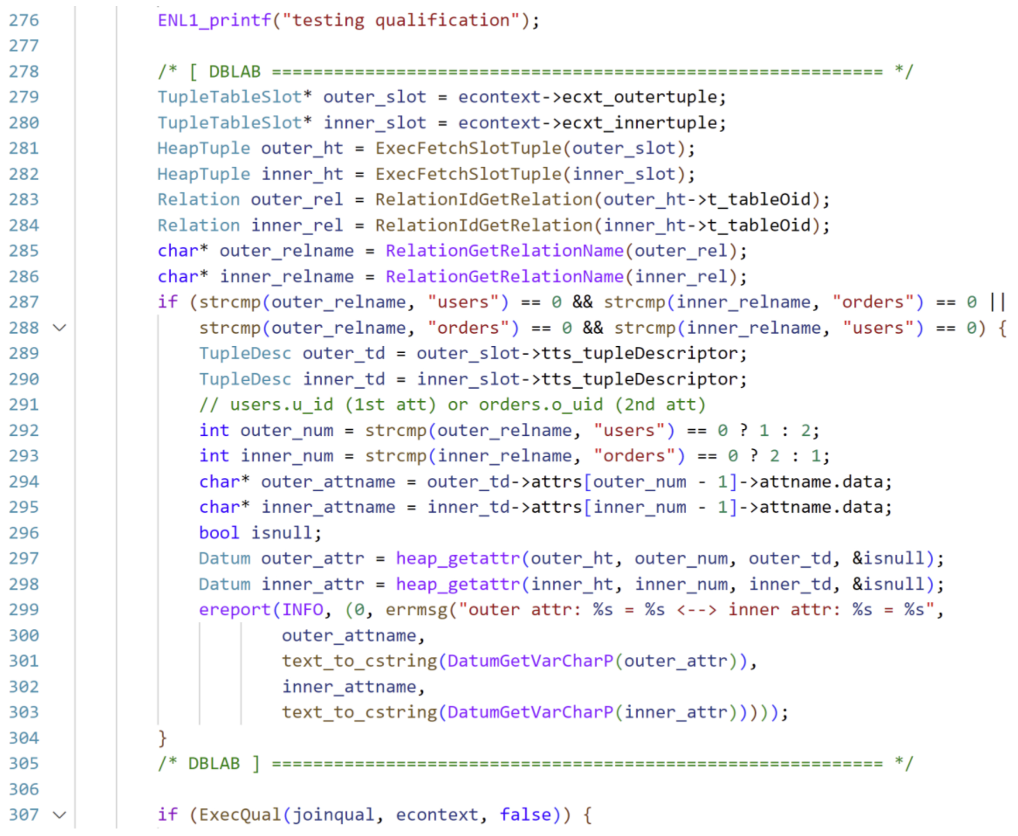
9.4.4 添加代码：输出嵌套循环算法的比较信息

通过在 ExecNestLoop 函数中添加代码，对内外层两个表的连接字段的值进行输出。

1. 在嵌套循环算法代码 ExecNestLoop 函数中，添加以下代码：

```
/* [ DBLAB ========================================================== */
TupleTableSlot* outer_slot = econtext->ecxt_outertuple;
TupleTableSlot* inner_slot = econtext->ecxt_innertuple;
HeapTuple outer_ht = ExecFetchSlotTuple(outer_slot);
HeapTuple inner_ht = ExecFetchSlotTuple(inner_slot);
Relation outer_rel = RelationIdGetRelation(outer_ht->t_tableOid);
Relation inner_rel = RelationIdGetRelation(inner_ht->t_tableOid);
char* outer_relname = RelationGetRelationName(outer_rel);
char* inner_relname = RelationGetRelationName(inner_rel);
if (strcmp(outer_relname, "users") == 0 && strcmp(inner_relname, "orders") == 0 ||
    strcmp(outer_relname, "orders") == 0 && strcmp(inner_relname, "users") == 0) {
    TupleDesc outer_td = outer_slot->tts_tupleDescriptor;
    TupleDesc inner_td = inner_slot->tts_tupleDescriptor;
    // users.u_id (1st att) or orders.o_uid (2nd att)
    int outer_num = strcmp(outer_relname, "users") == 0 ? 1 : 2;
    int inner_num = strcmp(inner_relname, "orders") == 0 ? 2 : 1;
    char* outer_attname = outer_td->attrs[outer_num - 1]->attname.data;
    char* inner_attname = inner_td->attrs[inner_num - 1]->attname.data;
    bool isnull;
    Datum outer_attr = heap_getattr(outer_ht, outer_num, outer_td, &isnull);
    Datum inner_attr = heap_getattr(inner_ht, inner_num, inner_td, &isnull);
    ereport(INFO, (0, errmsg("outer attr: %s = %s <--> inner attr: %s = %s",
            outer_attname,
            text_to_cstring(DatumGetVarCharP(outer_attr)),
            inner_attname,
            text_to_cstring(DatumGetVarCharP(inner_attr)))));
}
/* DBLAB ] ========================================================== */
```

添加代码的具体位置如图 9.3 所示。



```
ENL1_printf("testing qualification");

/* [ DBLAB ========================================================== */
TupleTableSlot* outer_slot = econtext->ecxt_outertuple;
TupleTableSlot* inner_slot = econtext->ecxt_innertuple;
HeapTuple outer_ht = ExecFetchSlotTuple(outer_slot);
HeapTuple inner_ht = ExecFetchSlotTuple(inner_slot);
Relation outer_rel = RelationIdGetRelation(outer_ht->t_tableOid);
Relation inner_rel = RelationIdGetRelation(inner_ht->t_tableOid);
char* outer_relname = RelationGetRelationName(outer_rel);
char* inner_relname = RelationGetRelationName(inner_rel);
if (strcmp(outer_relname, "users") == 0 && strcmp(inner_relname, "orders") == 0 ||
    strcmp(outer_relname, "orders") == 0 && strcmp(inner_relname, "users") == 0) {
    TupleDesc outer_td = outer_slot->tts_tupleDescriptor;
    TupleDesc inner_td = inner_slot->tts_tupleDescriptor;
    // users.u_id (1st att) or orders.o_uid (2nd att)
    int outer_num = strcmp(outer_relname, "users") == 0 ? 1 : 2;
    int inner_num = strcmp(inner_relname, "orders") == 0 ? 2 : 1;
    char* outer_attname = outer_td->attrs[outer_num - 1]->attname.data;
    char* inner_attname = inner_td->attrs[inner_num - 1]->attname.data;
    bool isnull;
    Datum outer_attr = heap_getattr(outer_ht, outer_num, outer_td, &isnull);
    Datum inner_attr = heap_getattr(inner_ht, inner_num, inner_td, &isnull);
    ereport(INFO, (0, errmsg("outer attr: %s = %s <--> inner attr: %s = %s",
            outer_attname,
            text_to_cstring(DatumGetVarCharP(outer_attr)),
            inner_attname,
            text_to_cstring(DatumGetVarCharP(inner_attr)))));
}
/* DBLAB ] ========================================================== */

if (ExecQual(joinqual, econtext, false)) {
```

图 9.3 在 ExecNestLoop 函数中添加代码的位置

对于添加的代码，解释如下：

- outer_slot 和 inner_slot 分别是两个连接表（外表和内表）的当前元组的 TupleTableSlot。
- 通过 ExecFetchSlotTuple 函数获得的 outer_ht 和 inner_ht 分别是外表和内表的当前元组的 HeapTuple。
- outer_rel 和 inner_rel 分别是外表和内表对应的 Relation 结构。
- outer_relname 和 inner_relname 分别是外表和内表的表名。通过 if 语句判断，只有当外表和内表的表名分别为 users 和 orders 或者 order 和 users 时，才执行下面的代码。
- outer_td 和 inner_td 分别是外表和内表对应的 TupleDesc。
- outer_num 和 inner_num 分别是外表和内表的当前元组的指定属性编号（从 1 开始）。对于 users 表为其第 1 个属性 u_id，对于 orders 表为其第 2 个属性 o_uid。
- 分别获取外表和内表指定属性名称到 outer_attname 和 inner_attname。
- 通过 heap_getattr 宏调用分别获取外表属性值 outer_attr 和内表属性值 inner_attr。
- 调用 ereport 和 errmsg 将外表和内表的对应属性名与属性值输出。

2. 停止 openGauss 数据库服务器：

```
[dblab@eduog openGauss-server-v3.0.0]$ gs_ctl stop -D $GAUSSHOME/data -Z single_node -l logfile
```

3. 重新编译并安装 openGauss:

```
[dblab@eduog openGauss-server-v3.0.0]$ make -j4
[dblab@eduog openGauss-server-v3.0.0]$ make install
```

4. gsql 连接数据库服务器：

```
[dblab@eduog openGauss-server-v3.0.0]$ gsql railway -r
gsql ((openGauss 3.0.0 build ) compiled at 2023-01-29 11:22:50 commit 0 last mr  debug)
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

railway=#
```

5. 重新设置优化器参数：

```
railway=# SET enable_hashjoin = off;
railway=# SET enable_mergejoin = off;
```

6. 再次执行连接查询：

```
railway=# SELECT u_id, u_name, o_id, o_uid, o_tid FROM users INNER JOIN orders ON u_id = o_uid;
INFO:  ExecNestLoop: getting info from node
INFO:  ExecNestLoop: entering main loop
INFO:  ExecNestLoop: getting new outer tuple
INFO:  ExecNestLoop: saving new outer tuple information
INFO:  ExecNestLoop: rescanning inner plan
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  outer attr: o_uid = 1 <--> inner attr: o_id = 1
INFO:  ExecNestLoop: qualification succeeded, projecting tuple
INFO:  ExecNestLoop: getting info from node
INFO:  ExecNestLoop: entering main loop
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: no inner tuple, need new outer tuple
INFO:  ExecNestLoop: getting new outer tuple
INFO:  ExecNestLoop: saving new outer tuple information
INFO:  ExecNestLoop: rescanning inner plan
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  outer attr: o_uid = 3 <--> inner attr: o_id = 3
INFO:  ExecNestLoop: qualification succeeded, projecting tuple
INFO:  ExecNestLoop: getting info from node
INFO:  ExecNestLoop: entering main loop
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: no inner tuple, need new outer tuple
INFO:  ExecNestLoop: getting new outer tuple
INFO:  ExecNestLoop: saving new outer tuple information
INFO:  ExecNestLoop: rescanning inner plan
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  outer attr: o_uid = 3 <--> inner attr: o_id = 3
INFO:  ExecNestLoop: qualification succeeded, projecting tuple
INFO:  ExecNestLoop: getting info from node
INFO:  ExecNestLoop: entering main loop
```

```
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: no inner tuple, need new outer tuple
INFO:  ExecNestLoop: getting new outer tuple
INFO:  ExecNestLoop: no outer tuple, ending join
 u_id | u_name | o_id | o_uid | o_tid
------+--------+------+-------+-------
 1    | 张三   |    1 | 1     | C2002
 3    | 王五   |    2 | 3     | G321
 3    | 王五   |    3 | 3     | G1709
(3 rows)
```

仔细观察输出信息，可以看到，以下 3 行输出信息，即对于外表 orders 表的每个元组，内表 users 表只进行了一次属性值比较操作（即每轮内层循环只执行了 1 次），而且在这一次比较操作中，外表和内表的对应属性值还恰好相等。这是什么原因呢？

回看上一步执行 EXPLAIN 语句输出的查询执行计划，对外表 orders 进行的是顺序扫描，而对内表 users 进行的是索引扫描。所谓索引扫描，就是用外表中当前元组的连接属性值作为键，到索引结构中进行值查找，由于该索引是建立在 users 表主键上的唯一索引，因而键查找会返回唯一值，即 users 表中 u_id 与 orders 表当前元组的 o_uid 值相等的唯一一个元组。这就解释了为什么对于外表的一个元组，内表仅进行一次比较操作，而且恰好对应属性值相等的原因了。有什么办法能够禁用索引扫描操作吗？

```
INFO:  outer attr: o_uid = 1 <--> inner attr: o_id = 1
INFO:  outer attr: o_uid = 3 <--> inner attr: o_id = 3
INFO:  outer attr: o_uid = 3 <--> inner attr: o_id = 3
```

7. 通过 SET 配置参数 enable_indexscan 为 off，禁止使用索引扫描。

```
railway=# SET enable_indexscan = off;
SET
```

8. 执行 EXPLAIN 语句，查看查询执行计划：

```
railway=# EXPLAIN SELECT u_id, u_name, o_id, o_uid, o_tid FROM users INNER JOIN orders ON u_id =
o_uid;
                             QUERY PLAN
--------------------------------------------------------------------
 Nested Loop  (cost=0.00..1231.77 rows=287 width=196)
   Join Filter: ((users.u_id)::text = (orders.o_uid)::text)
   ->  Seq Scan on orders  (cost=0.00..12.87 rows=287 width=100)
   ->  Materialize  (cost=0.00..14.20 rows=280 width=96)
         ->  Seq Scan on users  (cost=0.00..12.80 rows=280 width=96)
(5 rows)
```

对比此时查询执行计划与前面查询执行计划的区别。可以看到，orders 表和 users 表均为 Seq Scan 即顺序扫描操作。但在 users 表的 Seq Scan 上面，有一步 Materialize 即物化操作。

9. 通过 SET 配置参数 enable_material 为 off，禁止使用物化操作。

```
railway=# SET enable_material = off;
SET
```

10. 执行 EXPLAIN 语句，查看查询执行计划：

```
railway=# EXPLAIN SELECT u_id, u_name, o_id, o_uid, o_tid FROM users INNER JOIN orders ON u_id =
o_uid;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Nested Loop  (cost=0.29..1249.30 rows=287 width=196)
   ->  Seq Scan on orders  (cost=0.00..12.87 rows=287 width=100)
   ->  Bitmap Heap Scan on users  (cost=0.29..4.30 rows=1 width=96)
         Recheck Cond: ((u_id)::text = (orders.o_uid)::text)
         ->  Bitmap Index Scan on pk_users  (cost=0.00..0.29 rows=1 width=0)
               Index Cond: ((u_id)::text = (orders.o_uid)::text)
(6 rows)
```

发现这次查询执行计划改为使用位图索引扫描（Bitmap Index Scan）操作了。

11. 通过 SET 配置参数 enable_ bitmapscan 为 off，禁止使用位图扫描操作。

```
railway=# SET enable_bitmapscan = off;
SET
```

12. 执行 EXPLAIN 语句，查看查询执行计划：

```
railway=# EXPLAIN SELECT u_id, u_name, o_id, o_uid, o_tid FROM users INNER JOIN orders ON u_id =
o_uid;
                           QUERY PLAN
-----------------------------------------------------------------
 Nested Loop  (cost=0.00..4620.90 rows=287 width=196)
   Join Filter: ((users.u_id)::text = (orders.o_uid)::text)
   ->  Seq Scan on users  (cost=0.00..12.80 rows=280 width=96)
   ->  Seq Scan on orders  (cost=0.00..12.87 rows=287 width=100)
(4 rows)
```

观察发现，此时的查询执行计划中只有 Seq Scan 操作和 Join Filter 操作了。其中 Join Filter 就是两个连接属性值是否相等的条件判断。但是与之前使用 Index Scan 的查询执行计划相比，外表和内表发生了互换。这是 openGuass 的优化器做出的动态调整，用户无法自行改变。不过我们之前添加的代码是考虑了外表和内表交换的情况的，能够根据表名确定对应的属性编号。

13. 保证之前提及的优化器参数已禁用：

```
railway=#
SET enable_hashjoin = off;
SET enable_mergejoin = off;
SET enable_indexscan = off;
SET enable_material = off;
SET enable_bitmapscan = off;
```

14. 再次执行查询，得到输出信息：

```
railway=# SELECT u_id, u_name, o_id, o_uid, o_tid FROM users INNER JOIN orders ON u_id = o_uid;
INFO:  ExecNestLoop: getting info from node
INFO:  ExecNestLoop: entering main loop
INFO:  ExecNestLoop: getting new outer tuple
INFO:  ExecNestLoop: saving new outer tuple information
```

```
INFO:  ExecNestLoop: rescanning inner plan
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  outer attr: u_id = 1 <--> inner attr: o_uid = 1
INFO:  ExecNestLoop: qualification succeeded, projecting tuple
INFO:  ExecNestLoop: getting info from node
INFO:  ExecNestLoop: entering main loop
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  outer attr: u_id = 1 <--> inner attr: o_uid = 3
INFO:  ExecNestLoop: qualification failed, looping
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  outer attr: u_id = 1 <--> inner attr: o_uid = 3
INFO:  ExecNestLoop: qualification failed, looping
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: no inner tuple, need new outer tuple
INFO:  ExecNestLoop: getting new outer tuple
INFO:  ExecNestLoop: saving new outer tuple information
INFO:  ExecNestLoop: rescanning inner plan
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  outer attr: u_id = 2 <--> inner attr: o_uid = 1
INFO:  ExecNestLoop: qualification failed, looping
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  outer attr: u_id = 2 <--> inner attr: o_uid = 3
INFO:  ExecNestLoop: qualification failed, looping
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  outer attr: u_id = 2 <--> inner attr: o_uid = 3
INFO:  ExecNestLoop: qualification failed, looping
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: no inner tuple, need new outer tuple
INFO:  ExecNestLoop: getting new outer tuple
INFO:  ExecNestLoop: saving new outer tuple information
INFO:  ExecNestLoop: rescanning inner plan
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  outer attr: u_id = 3 <--> inner attr: o_uid = 1
INFO:  ExecNestLoop: qualification failed, looping
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  outer attr: u_id = 3 <--> inner attr: o_uid = 3
```

```
INFO:  ExecNestLoop: qualification succeeded, projecting tuple
INFO:  ExecNestLoop: getting info from node
INFO:  ExecNestLoop: entering main loop
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: testing qualification
INFO:  outer attr: u_id = 3 <--> inner attr: o_uid = 3
INFO:  ExecNestLoop: qualification succeeded, projecting tuple
INFO:  ExecNestLoop: getting info from node
INFO:  ExecNestLoop: entering main loop
INFO:  ExecNestLoop: getting new inner tuple
INFO:  ExecNestLoop: no inner tuple, need new outer tuple
INFO:  ExecNestLoop: getting new outer tuple
INFO:  ExecNestLoop: no outer tuple, ending join
 u_id | u_name | o_id | o_uid | o_tid
------+--------+------+-------+-------
 1    | 张三   |    1 | 1     | C2002
 3    | 王五   |    2 | 3     | G321
 3    | 王五   |    3 | 3     | G1709
(3 rows)
```

此次，我们终于可以看到了，该查询按照嵌套循环算法的流程进行了连接属性值的比较：

```
INFO:  outer attr: u_id = 1 <--> inner attr: o_uid = 1
INFO:  outer attr: u_id = 1 <--> inner attr: o_uid = 3
INFO:  outer attr: u_id = 1 <--> inner attr: o_uid = 3
INFO:  outer attr: u_id = 2 <--> inner attr: o_uid = 1
INFO:  outer attr: u_id = 2 <--> inner attr: o_uid = 3
INFO:  outer attr: u_id = 2 <--> inner attr: o_uid = 3
INFO:  outer attr: u_id = 3 <--> inner attr: o_uid = 1
INFO:  outer attr: u_id = 3 <--> inner attr: o_uid = 3
INFO:  outer attr: u_id = 3 <--> inner attr: o_uid = 3
```

观察输出信息，就是按照嵌套循环算法步骤，外表 users 的当前元组与内表 orders 的当前元组进行了 9 次比较，即对于 users 表的每行，orders 表进行一轮遍历，每轮遍历进行 3 次比较。

至此，从实验的角度验证了嵌套循环算法原理。

如果在执行上述连接查询 SQL 语句时数据库服务器崩溃或 gsql 连接断开，或没有实现预期的输出效果，是由于编写代码过程中引入了错误（bug）。此时，可通过使用第 3 章中已实践的单步调试方法对添加代码进行调试（debug），以排除错误。通过“编辑——编译——测试——调试”这样的步骤循环，直到成功执行为止。

9.5 实验结果

【请按照要求完成实验操作】

1. 完成实验步骤 9.4.1 节、9.4.2 节和 9.4.3 节，输出 ExecNestLoop 函数中的调试信息。

2. 请解释说明在实验步骤 9.4.4 节中，为什么在配置 enable_indexscan、enable_material 和 enable_ bitmapscan 参数为 off 之前连接属性值比较信息输出为 3 行；而之后连接属性值比较信息输出变为 9 行。

3. 完成实验步骤 9.4.4 节，输出 9 行连接属性值比较信息。

9.6 讨论与总结

【请将实验中遇到的问题描述、解决办法与思考讨论列在下面，并对本实验进行总结。】