Hash Join 算子的实现

1. 数据生成

1.1 建表 （如果之前存在t1、t2 先删除表 drop table t1,t2; 或者使用其他表名）

create table t1 (c1 int primary key, c2 int);

create table t2 (c1 int primary key, c2 int);

1.2 插入测试数据

insert into t1 SELECT generate_series(1,10000) as key, (random()*(10^3))::integer;

insert into t2 SELECT generate_series(1,10000) as key, (random()*(10^3))::integer;

1.3 Hash Join 执行

例如：explain select * from t1,t2 where t1.c1<5 and t1.c2 = t2.c2;

```
postgres=# explain select * from t1,t2 where t1.c1<5 and t1.c2 = t2.c2;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=8.41..228.81 rows=40 width=16)
   Hash Cond: (t2.c2 = t1.c2)
   ->  Seq Scan on t2  (cost=0.00..145.00 rows=10000 width=8)
   ->  Hash  (cost=8.36..8.36 rows=4 width=8)
         ->  Index Scan using t1_pkey on t1  (cost=0.29..8.36 rows=4 width=8)
               Index Cond: (c1 < 5)
(6 rows)
```

lefttree对应的是SeqScan,

righttree对应的是Hash,

即左树(outer relation)为t2的顺序扫描运算生成的relation,

右树(inner relation)为t1的索引扫描（Index Scan)运算生成的relation(在此relation上创建Hash Table)

2. 回顾 计划树

Hash Join 计划树PlanTree

2.1 log中的计划树

```
:planTree
   {HASHJOIN
   :startup_cost 60.85
   :total_cost 124.53
   :plan_rows 2260
   :plan_width 8
   :parallel_aware false
   :plan_node_id 0
   :targetlist (
      {TARGETENTRY
      :expr
         {VAR
         :varno 65001
         :varattno 1
         :vartype 23
         :vartypmod -1
         :varcollid 0
         :varlevelsup 0
         :varnoold 1
         :varoattno 1
         :location 7
         }
      :resno 1
      :resname c1
      :ressortgroupref 0
      :resorigtbl 16390
      :resorigcol 1
      :resjunk false
      }
      {TARGETENTRY
      :expr
         {VAR
         :varno 65000
         :varattno 1
         :vartype 23
         :vartypmod -1
         :varcollid 0
         :varlevelsup 0
         :varnoold 2
         :varoattno 2
         :location 14
         }
      :resno 2
      :resname c2
      :ressortgroupref 0
      :resorigtbl 16385
      :resorigcol 2
      :resjunk false
      }
   )
   :qual <>
```

```
:lefttree
   {SEQSCAN
   :startup_cost 0.00
   :total_cost 32.60
   :plan_rows 2260
   :plan_width 4
   :parallel_aware false
   :plan_node_id 1
   :targetlist (
      {TARGETENTRY
      :expr
         {VAR
         :varno 1
         :varattno 1
         :vartype 23
         :vartypmod -1
         :varcollid 0
         :varlevelsup 0
         :varnoold 1
         :varoattno 1
         :location -1
         }
      :resno 1
      :resname <>
      :ressortgroupref 0
      :resorigtbl 0
      :resorigcol 0
      :resjunk false
      }
      {TARGETENTRY
      :expr
         {VAR
         :varno 1
         :varattno 2
         :vartype 23
         :vartypmod -1
         :varcollid 0
         :varlevelsup 0
         :varnoold 1
         :varoattno 2
         :location -1
         }
      :resno 2
      :resname <>
      :ressortgroupref 0
      :resorigtbl 0
      :resorigcol 0
      :resjunk false
      }
   )
   :qual <>
   :lefttree <>
   :righttree <>
   :initPlan <>
   :extParam (b)
   :allParam (b)
   :scanrelid 1
   }
```

```
:lefttree
   {SEQSCAN
   :startup_cost 0.00
   :total_cost 32.60
   :plan_rows 2260
   :plan_width 4
   :parallel_aware false
   :plan_node_id 1
   :targetlist (
      {TARGETENTRY
      :expr
         {VAR
         :varno 1
         :varattno 1
         :vartype 23
         :vartypmod -1
         :varcollid 0
         :varlevelsup 0
         :varnoold 1
         :varoattno 1
         :location -1
         }
      :resno 1
      :resname <>
      :ressortgroupref 0
      :resorigtbl 0
      :resorigcol 0
      :resjunk false
      }
      {TARGETENTRY
      :expr
         {VAR
         :varno 1
         :varattno 2
         :vartype 23
         :vartypmod -1
         :varcollid 0
         :varlevelsup 0
         :varnoold 1
         :varoattno 2
         :location -1
         }
      :resno 2
      :resname <>
      :ressortgroupref 0
      :resorigtbl 0
      :resorigcol 0
      :resjunk false
      }
   )
   :qual <>
   :lefttree <>
   :righttree <>
   :initPlan <>
   :extParam (b)
   :allParam (b)
   :scanrelid 1
   }
```

2.2 GDB Hash Join 计划树

启动gdb,设置断点,进入ExecHashJoin

```
#13 0x00000000006b3a78 in main (argc=3, argv=0xf774d0) at main.c:2
(gdb) b ExecHashJoin
Breakpoint 1 at 0x67dea1: file nodeHashjoin.c, line 63.
(gdb)
```

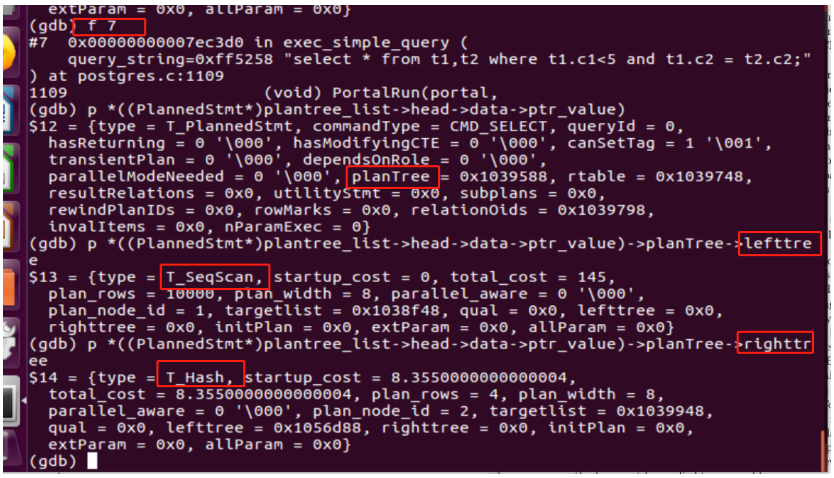
执行SQL select * from t1,t2 where t1.c1<5 and t1.c2 = t2.c2;

继续执行,进入Hash Join

```
#13 0x00000000006b3a78 in main (argc=3, argv=0xf774d0) at main.c:228
(gdb) b ExecHashJoin
Breakpoint 1 at 0x67dea1: file nodeHashjoin.c, line 63.
(gdb) c
Continuing.

Breakpoint 1, ExecHashJoin (node=0x1059908) at nodeHashjoin.c:63
63      {
(gdb) bt
#0  ExecHashJoin (node=0x1059908) at nodeHashjoin.c:63
#1  0x0000000000661a4d in ExecProcNode (node=0x1059908) at execProcnode.c:484
#2  0x000000000065dd51 in ExecutePlan (estate=0x10597f8, planstate=0x1059908,
    use_parallel_mode=0 '\000', operation=CMD_SELECT, sendTuples=1 '\001',
    numberTuples=0, direction=ForwardScanDirection, dest=0x103a088)
    at execMain.c:1567
#3  0x000000000065c07b in standard_ExecutorRun (queryDesc=0x104e548,
    direction=ForwardScanDirection, count=0) at execMain.c:339
#4  0x000000000065bf70 in ExecutorRun (queryDesc=0x104e548,
    direction=ForwardScanDirection, count=0) at execMain.c:287
#5  0x00000000007f27a9 in PortalRunSelect (portal=0x1050ed8, forward=1 '\001',
    count=0, dest=0x103a088) at pquery.c:948
```

第七帧 调试左右子树节点



3 Hash Join 算子实现（执行过程）

3.1 数据结构

JoinState

Hash/NestLoop/Merge Join的基类

```
/* ----------------
 * JoinState information
 *
 * Superclass for state nodes of join plans.
 * Hash/NestLoop/Merge Join的基类
 * ----------------
 */
typedef struct JoinState
{
 PlanState ps;//基类PlanState
 JoinType jointype;//连接类型
 //在找到一个匹配inner tuple的时候,如需要跳转到下一个outer tuple,则该值为T
 bool single_match; /* True if we should skip to next outer tuple
 * after finding one inner match */
 //连接条件表达式(除了ps.qual)
 ExprState *joinqual; /* JOIN quals (in addition to ps.qual) */
} JoinState;
```

HashJoinState

Hash Join运行期状态结构体

```
/* these structs are defined in executor/hashjoin.h: */
typedef struct HashJoinTupleData *HashJoinTuple;
typedef struct HashJoinTableData *HashJoinTable;

typedef struct HashJoinState
{
 JoinState js; /* 基类;its first field is NodeTag */
 ExprState *hashclauses;//hash连接条件
 List *hj_OuterHashKeys; /* 外表条件链表;list of ExprState nodes */
 List *hj_InnerHashKeys; /* 内表连接条件;list of ExprState nodes */
 List *hj_HashOperators; /* 操作符OIDs链表;list of operator OIDs */
 HashJoinTable hj_HashTable;//Hash表
 uint32 hj_CurHashValue;//当前的Hash值
 int hj_CurBucketNo;//当前的bucket编号
 int hj_CurSkewBucketNo;//行倾斜bucket编号
 HashJoinTuple hj_CurTuple;//当前元组
 TupleTableSlot *hj_OuterTupleSlot;//outer relation slot
 TupleTableSlot *hj_HashTupleSlot;//Hash tuple slot
 TupleTableSlot *hj_NullOuterTupleSlot;//用于外连接的outer虚拟slot
```

```
 TupleTableSlot *hj_NullInnerTupleSlot;//用于外连接的inner虚拟slot
 TupleTableSlot *hj_FirstOuterTupleSlot;//
 int hj_JoinState;//JoinState状态
 bool hj_MatchedOuter;//是否匹配
 bool hj_OuterNotEmpty;//outer relation是否为空
} HashJoinState;
```

3.2 源码解读

ExecHashJoinImpl函数把Hash Join划分为多个阶段/状态(有限状态机),保存在HashJoinState->**hj_JoinState**字段中,这些状态分别是分别为

HJ_BUILD_HASHTABLE/HJ_NEED_NEW_OUTER/HJ_SCAN_BUCKET/HJ_FILL_OUTER_TUPLE/HJ_FILL_INNER_TUPLES/HJ_NEED_NEW

含义分别为:

1. HJ_BUILD_HASHTABLE:创建Hash表;
2. HJ_NEED_NEW_OUTER:扫描outer relation,计算外表连接键的hash值,把相匹配元组放在合适的bucket中;
3. HJ_SCAN_BUCKET:扫描bucket,匹配的tuple返回
4. HJ_FILL_OUTER_TUPLE:当前outer relation元组已耗尽，因此检查是否发出一个虚拟的外连接元组。
5. HJ_FILL_INNER_TUPLES:已完成一个批处理，但做的是右外连接/完全连接,填充虚拟连接元组
6. HJ_NEED_NEW_BATCH:开启下一批次

注意:在work_mem不足以装下Hash Table时,分批执行.每个批次执行时,会把outer relation与inner relation匹配(指hash值一样)的tuple会存储起来,放在合适的批次文件中(hashtable->outerBatchFile[batchno]),以避免多次的outer relation扫描

```
/* ----------------------------------------------------------------
 * ExecHashJoin
 *
 * This function implements the Hybrid Hashjoin algorithm.
 *这个函数实现了混合Hash Join算法
 * Note: the relation we build hash table on is the "inner"
 * the other one is "outer".
 *注意:在inner上创建hash表,另外一个参与连接的成为outer
 * ----------------------------------------------------------------
 */
TupleTableSlot * /* return: a tuple or NULL */
ExecHashJoin(HashJoinState *node)
{
 PlanState *outerNode;
 HashState *hashNode;
 List *joinqual;
 List *otherqual;
 ExprContext *econtext;
 ExprDoneCond isDone;
 HashJoinTable hashtable;
 TupleTableSlot *outerTupleSlot;
 uint32 hashvalue;
 int batchno;

 /*
 * get information from HashJoin node
 * 从HashJon Node中获取信息
 */
 joinqual = node->js.joinqual;
 otherqual = node->js.ps.qual;
 hashNode = (HashState *) innerPlanState(node);
 outerNode = outerPlanState(node);
 hashtable = node->hj_HashTable;
 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)
{
TupleTableSlot *result;

result = ExecProject(node->js.ps.ps_ProjInfo, &isDone);
if (isDone == 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.
 * 重置每个元组内存上下文以释放在前一个元组处理周期中分配的所有表达式计算存储。
 */
ResetExprContext(econtext);

/*
 * run the hash join state machine
 * 执行hash join状态机
 */
for (;;)
{
switch (node->hj_JoinState)
{
case HJ_BUILD_HASHTABLE:

/*
 * First time through: build hash table for inner relation.
 * 第一次的处理逻辑:为inner relation建立hash表
 */
Assert(hashtable == NULL);

/*
 * If the outer relation is completely empty, and it's not
 * right/full join, we can quit without building the hash
 * table. However, for an inner join it is only a win to
 * check this when the outer relation's startup cost is less
 * than the projected cost of building the hash table.
 * Otherwise it's best to build the hash table first and see
 * if the inner relation is empty. (When it's a left join, we
 * should always make this check, since we aren't going to be
 * able to skip the join on the strength of an empty inner
 * relation anyway.)
 * 如果外部关系是空的，并且它不是右外/完全连接，可以在不构建哈希表的情况下退出。
 * 但是，对于内连接，只有当外部关系的启动成本小于构建哈希表的预期成本时，才需要检查这一点。
 * 否则，最好先构建哈希表，看看内部关系是否为空。
 * (当它是左外连接时，应该始终进行检查，因为无论如何，都不能基于空的内部关系跳过连接。)
 *
 *
 * If we are rescanning the join, we make use of information
 * gained on the previous scan: don't bother to try the
 * prefetch if the previous scan found the outer relation
 * nonempty. This is not 100% reliable since with new
 * parameters the outer relation might yield different
 * results, but it's a good heuristic.
```

```
 * 如果需要重新扫描连接，将利用上次扫描结果中获得的信息：
 * 如果上次扫描发现外部关系非空，则不必尝试预取。
 * 但这不是100%可靠的，因为有了新的参数，外部关系可能会产生不同的结果，但这是一个很好的启发式。
 *
 * The only way to make the check is to try to fetch a tuple
 * from the outer plan node.  If we succeed, we have to stash
 * it away for later consumption by ExecHashJoinOuterGetTuple.
 * 进行检查的唯一方法是从外部plan节点获取一个元组。
 * 如果成功了，就必须通过ExecHashJoinOuterGetTuple将其存储起来，以便以后使用。
 */
if (HJ_FILL_INNER(node))
{
/* no chance to not build the hash table */
//不构建哈希表是不可能的了
node->hj_FirstOuterTupleSlot = NULL;
}
else if (HJ_FILL_OUTER(node) ||
(outerNode->plan->startup_cost < hashNode->ps.plan->total_cost &&
!node->hj_OuterNotEmpty))
{
node->hj_FirstOuterTupleSlot = ExecProcNode(outerNode);
if (TupIsNull(node->hj_FirstOuterTupleSlot))
{
node->hj_OuterNotEmpty = false;
return NULL;
}
else
node->hj_OuterNotEmpty = true;
}
else
node->hj_FirstOuterTupleSlot = NULL;

/*
 * create the hash table
 */
hashtable = ExecHashTableCreate((Hash *) hashNode->ps.plan,
node->hj_HashOperators,
HJ_FILL_INNER(node));
node->hj_HashTable = hashtable;

/*
 * execute the Hash node, to build the hash table
 * 创建哈希表。
 */
hashNode->hashtable = hashtable;
(void) MultiExecProcNode((PlanState *) hashNode);

/*
 * If the inner relation is completely empty, and we're not
 * doing a left outer join, we can quit without scanning the
 * outer relation.
 * 如果内部关系是空的，并且没有执行左外连接，可以在不扫描外部关系的情况下退出。
 */
if (hashtable->totalTuples == 0 && !HJ_FILL_OUTER(node))
return NULL;

/*
 * need to remember whether nbatch has increased since we
 * began scanning the outer relation
 * 需要记住自开始扫描外部关系以来nbatch是否增加了
```

```
*/
hashtable->nbatch_outstart = hashtable->nbatch;

/*
* Reset OuterNotEmpty for scan. (It's OK if we fetched a
* tuple above, because ExecHashJoinOuterGetTuple will
* immediately set it again.)
* 扫描前重置OuterNotEmpty。
* (在其上获取一个tuple是可以的，因为ExecHashJoinOuterGetTuple将立即再次设置它。)
*/
node->hj_OuterNotEmpty = false;

node->hj_JoinState = HJ_NEED_NEW_OUTER; //进入下一个状态

/* FALL THRU */

case HJ_NEED_NEW_OUTER: //扫描outer relation,计算外表连接键的hash值,把相匹配元组放在合适的bucket中

/*
* We don't have an outer tuple, try to get the next one
* 没有外部元组，试着获取下一个
*/
outerTupleSlot = ExecHashJoinOuterGetTuple(outerNode,
node,
&hashvalue);
if (TupIsNull(outerTupleSlot))
{
//如outerTupleSlot为NULL
/* end of batch, or maybe whole join */
//完成此批数据处理,或者可能是全连接
if (HJ_FILL_INNER(node))
{
/* set up to scan for unmatched inner tuples */
//不匹配的行,填充NULL(外连接)
ExecPrepHashTableForUnmatched(node);
node->hj_JoinState = HJ_FILL_INNER_TUPLES;
}
else
node->hj_JoinState = HJ_NEED_NEW_BATCH;
continue;
}
//设置变量
econtext->ecxt_outertuple = outerTupleSlot;
node->hj_MatchedOuter = false;

/*
* Find the corresponding bucket for this tuple in the main
* hash table or skew hash table.
* 在主哈希表或斜哈希表中为这个元组找到对应的bucket。
*/
node->hj_CurHashValue = hashvalue;
//获取Hash Bucket并处理此批次
ExecHashGetBucketAndBatch(hashtable, hashvalue,
&node->hj_CurBucketNo, &batchno);
//Hash倾斜优化(某个值的数据特别多)
node->hj_CurSkewBucketNo = ExecHashGetSkewBucket(hashtable,
hashvalue);
node->hj_CurTuple = NULL;

/*
* The tuple might not belong to the current batch (where
```

```
 * "current batch" includes the skew buckets if any).
 * 元组可能不属于当前批处理(其中“当前批处理”包括倾斜桶-如果有的话)。
 */
if (batchno != hashtable->curbatch &&
node->hj_CurSkewBucketNo == INVALID_SKEW_BUCKET_NO)
{
/*
 * Need to postpone this outer tuple to a later batch.
 * Save it in the corresponding outer-batch file.
 * 需要将这个外部元组推迟到稍后的批处理。保存在相应的外部批处理文件中。
 * 也就是说,INNER和OUTER属于此批次的数据都可能存储在外存中
 */
Assert(batchno > hashtable->curbatch);
ExecHashJoinSaveTuple(ExecFetchSlotMinimalTuple(outerTupleSlot),
hashvalue,
&hashtable->outerBatchFile[batchno]);
/* Loop around, staying in HJ_NEED_NEW_OUTER state */
//循环，保持HJ_NEED_NEW_OUTER状态
continue;
}

/* OK, let's scan the bucket for matches */
//已完成此阶段,切换至HJ_SCAN_BUCKET状态
node->hj_JoinState = HJ_SCAN_BUCKET;

/* FALL THRU */

case HJ_SCAN_BUCKET:

/*
 * We check for interrupts here because this corresponds to
 * where we'd fetch a row from a child plan node in other join
 * types.
 */
CHECK_FOR_INTERRUPTS();

/*
 * Scan the selected hash bucket for matches to current outer
 * 扫描选定的散列桶，查找与当前外部匹配的散列桶
 */
if (!ExecScanHashBucket(node, econtext))
{
/* out of matches; check for possible outer-join fill */
node->hj_JoinState = HJ_FILL_OUTER_TUPLE;
continue;
}

/*
 * We've got a match, but still need to test non-hashed quals.
 * ExecScanHashBucket already set up all the state needed to
 * call ExecQual.
 * 发现一个匹配，但仍然需要测试非散列的quals。
 * ExecScanHashBucket已经设置了调用ExecQual所需的所有状态。
 * If we pass the qual, then save state for next call and have
 * ExecProject form the projection, store it in the tuple
 * table, and return the slot.
 * 如果我们传递了qual，那么将状态保存为下一次调用，
 * 并让ExecProject形成投影，将其存储在tuple table中，并返回slot。
 *
 * Only the joinquals determine tuple match status, but all
 * quals must pass to actually return the tuple.
```

```
* 只有连接条件joinquals确定元组匹配状态，但所有条件quals必须通过才能返回元组。
*/
if (joinqual == NIL || ExecQual(joinqual, econtext, false))
{
node->hj_MatchedOuter = true;
HeapTupleHeaderSetMatch(HJTUPLE_MINTUPLE(node->hj_CurTuple));

/* In an antijoin, we never return a matched tuple */
//反连接,则不能返回匹配的元组
if (node->js.jointype == JOIN_ANTI)
{
node->hj_JoinState = HJ_NEED_NEW_OUTER;
continue;
}

/*
* In a semijoin, we'll consider returning the first
* match, but after that we're done with this outer tuple.
* 如果只需要连接到第一个匹配的内表元组，那么可以考虑返回这个元组，
* 但是在此之后可以继续使用下一个外表元组。
*/
if (node->js.jointype == JOIN_SEMI)
node->hj_JoinState = HJ_NEED_NEW_OUTER;

if (otherqual == NIL ||
ExecQual(otherqual, econtext, false))
{
TupleTableSlot *result;
//执行投影操作
result = ExecProject(node->js.ps.ps_ProjInfo, &isDone);

if (isDone != ExprEndResult)
{
node->js.ps.ps_TupFromTlist =
(isDone == ExprMultipleResult);
return result;
}
}
else
InstrCountFiltered2(node, 1);//其他条件不匹配
}
else
InstrCountFiltered1(node, 1);//连接条件不匹配
break;

case HJ_FILL_OUTER_TUPLE:

/*
* The current outer tuple has run out of matches, so check
* whether to emit a dummy outer-join tuple. Whether we emit
* one or not, the next state is NEED_NEW_OUTER.
* 当前外部元组已耗尽匹配项，因此检查是否发出一个虚拟的外连接元组。
* 不管是否发出一个，下一个状态是NEED_NEW_OUTER。
*/
node->hj_JoinState = HJ_NEED_NEW_OUTER;

if (!node->hj_MatchedOuter &&
HJ_FILL_OUTER(node))
{
/*
* Generate a fake join tuple with nulls for the inner
```

```
 * tuple, and return it if it passes the non-join quals.
 * 为内部元组生成一个带有null的假连接元组，并在满足非连接条件quals时返回它。
 */
econtext->ecxt_innertuple = node->hj_NullInnerTupleSlot;

if (otherqual == NIL ||
ExecQual(otherqual, econtext, false))
{
TupleTableSlot *result;

result = ExecProject(node->js.ps.ps_ProjInfo, &isDone);

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

case HJ_FILL_INNER_TUPLES:

/*
 * We have finished a batch, but we are doing right/full join,
 * so any unmatched inner tuples in the hashtable have to be
 * emitted before we continue to the next batch.
 * 已经完成了一个批处理，但是做的是右外/完全连接，
 * 所以必须在继续下一个批处理之前发出散列表中任何不匹配的内部元组。
 */
if (!ExecScanHashTableForUnmatched(node, econtext))
{
/* no more unmatched tuples */
//不存在更多不匹配的元组,切换状态为HJ_NEED_NEW_BATCH(开始下一批次)
node->hj_JoinState = HJ_NEED_NEW_BATCH;
continue;
}

/*
 * Generate a fake join tuple with nulls for the outer tuple,
 * and return it if it passes the non-join quals.
 * 为外表元组生成一个带有null的假连接元组，并在满足非连接条件quals时返回它。
 */
econtext->ecxt_outertuple = node->hj_NullOuterTupleSlot;

if (otherqual == NIL ||
ExecQual(otherqual, econtext, false))
{
TupleTableSlot *result;

result = ExecProject(node->js.ps.ps_ProjInfo, &isDone);

if (isDone != ExprEndResult)
{
node->js.ps.ps_TupFromTlist =
(isDone == ExprMultipleResult);
return result;
}
```

```
403   }
404   else
405   InstrCountFiltered2(node, 1);
406   break;
407
408   case HJ_NEED_NEW_BATCH:
409
410   /*
411   * Try to advance to next batch. Done if there are no more.
412   * 尽量提前到下一批。如果没有了，就结束。
413   */
414   if (!ExecHashJoinNewBatch(node))
415   return NULL; /* end of join */
416   node->hj_JoinState = HJ_NEED_NEW_OUTER;//切换状态
417   break;
418
419   default: //非法的JoinState
420   elog(ERROR, "unrecognized hashjoin state: %d",
421   (int) node->hj_JoinState);
422   }
423   }
424 }
```

3.3 调试

启动gdb,设置断点,进入ExecHashJoin

```
#13 0x00000000006b3a78 in main (argc=3, argv=0xf774d0) at main.c:2
(gdb) b ExecHashJoin
Breakpoint 1 at 0x67dea1: file nodeHashjoin.c, line 63.
(gdb)
```

执行SQL select * from t1,t2 where t1.c1<5 and t1.c2 = t2.c2;

继续执行,进入Hash Join

```
#13 0x00000000006b3a78 in main (argc=3, argv=0xf774d0) at main.c:228
(gdb) b ExecHashJoin
Breakpoint 1 at 0x67dea1: file nodeHashjoin.c, line 63.
(gdb) c
Continuing.

Breakpoint 1, ExecHashJoin (node=0x1059908) at nodeHashjoin.c:63
63      {
(gdb) bt
#0  ExecHashJoin (node=0x1059908) at nodeHashjoin.c:63
#1  0x0000000000661a4d in ExecProcNode (node=0x1059908) at execProcnode.c:484
#2  0x000000000065dd51 in ExecutePlan (estate=0x10597f8, planstate=0x1059908,
    use_parallel_mode=0 '\000', operation=CMD_SELECT, sendTuples=1 '\001',
    numberTuples=0, direction=ForwardScanDirection, dest=0x103a088)
    at execMain.c:1567
#3  0x000000000065c07b in standard_ExecutorRun (queryDesc=0x104e548,
    direction=ForwardScanDirection, count=0) at execMain.c:339
#4  0x000000000065bf70 in ExecutorRun (queryDesc=0x104e548,
    direction=ForwardScanDirection, count=0) at execMain.c:287
#5  0x00000000007f27a9 in PortalRunSelect (portal=0x1050ed8, forward=1 '\001',
    count=0, dest=0x103a088) at pquery.c:948
```

第一帧 查看HashJoinState等变量值

```
(gdb) f 0
#0  ExecHashJoin (node=0x1059908) at nodeHashjoin.c:79
79              otherqual = node->js.ps.qual;
(gdb) l
74
75              /*
76               * get information from HashJoin node
77               */
78              joinqual = node->js.joinqual;
79              otherqual = node->js.ps.qual;
80              hashNode = (HashState *) innerPlanState(node);
81              outerNode = outerPlanState(node);
82              hashtable = node->hj_HashTable;
83              econtext = node->js.ps.ps_ExprContext;
(gdb) n
80              hashNode = (HashState *) innerPlanState(node);
(gdb)
81              outerNode = outerPlanState(node);
(gdb)
82              hashtable = node->hj_HashTable;
(gdb)
83              econtext = node->js.ps.ps_ExprContext;
(gdb)
90              if (node->js.ps.ps_TupFromTlist)
(gdb) p *node
$31 = {js = {ps = {type = T_HashJoinState, plan = 0x1039588,
      state = 0x10597f8, instrument = 0x0, worker_instrument = 0x0,
      targetlist = 0x1059b78, qual = 0x0, lefttree = 0x105a658,
      righttree = 0x105ad58, initPlan = 0x0, subPlan = 0x0, chgParam = 0x0,
      ps_ResultTupleSlot = 0x105c7a0, ps_ExprContext = 0x1059a18,
```

```
(gdb) p *hashNode
$32 = {ps = {type = T_HashState, plan = 0x10394f8, state = 0x10597f8,
    instrument = 0x0, worker_instrument = 0x0, targetlist = 0x105b048,
    qual = 0x0, lefttree = 0x105b0f8, righttree = 0x0, initPlan = 0x0,
    subPlan = 0x0, chgParam = 0x0, ps_ResultTupleSlot = 0x105af18,
    ps_ExprContext = 0x105ae68, ps_ProjInfo = 0x0,
    ps_TupFromTlist = 0 '\000'}, hashtable = 0x0, hashkeys = 0x105ce90}
(gdb) p *hashtable
Cannot access memory at address 0x0
```

HJ_BUILD_HASHTABLE->执行相关判断,本例为内连接,因此不存在FILL_OUTER等情况

```
$33 = 1
(gdb) n
151                                                     (outerNode->plan->start
up_cost < hashNode->ps.plan->total_cost &&
(gdb) n
150                                 else if (HJ_FILL_OUTER(node) ||
(gdb) n
152                                                      !node->hj_OuterNotEmpt
y))
(gdb) n
151                                                     (outerNode->plan->start
up_cost < hashNode->ps.plan->total_cost &&
(gdb) n
154                                         node->hj_FirstOuterTupleSlot = E
xecProcNode(outerNode);
(gdb)
155                                         if (TupIsNull(node->hj_FirstOute
rTupleSlot))
(gdb)
161                                                 node->hj_OuterNotEmpty =
 true;
(gdb)
155                                         if (TupIsNull(node->hj_FirstOute
rTupleSlot))
(gdb)
171                                                                       H
J_FILL_INNER(node));
(gdb)
169                                 hashtable = ExecHashTableCreate((Hash *)
 hashNode->ps.plan,
```

HJ_BUILD_HASHTABLE->outer node的启动成本低于创建Hash表的总成本而且outer relation为空(初始化node->hj_OuterNotEmpty为false),那么尝试获取outer relation的第一个元组,如为NULL,则可快速返回NULL,否则设置node->hj_OuterNotEmpty标记为T

HJ_BUILD_HASHTABLE->使用的Hash函数

```
(gdb) p *hashtable->inner_hashfunctions
$15 = {fn_addr = 0x4c8a0a <hashtext>, fn_oid = 400, fn_nargs = 1, fn_strict = true, fn_retset = false, fn_stats = 2 '\002',
  fn_extra = 0x0, fn_mcxt = 0x3053a50, fn_expr = 0x0}
(gdb) p *hashtable->outer_hashfunctions
$16 = {fn_addr = 0x4c8a0a <hashtext>, fn_oid = 400, fn_nargs = 1, fn_strict = true, fn_retset = false, fn_stats = 2 '\002',
  fn_extra = 0x0, fn_mcxt = 0x3053a50, fn_expr = 0x0}
```

HJ_BUILD_HASHTABLE->批次数为1,只需要执行1个批次即可

```
(gdb) n
304 hashtable->nbatch_outstart = hashtable->nbatch;
(gdb) p hashtable->nbatch
$19 = 1
```

HJ_BUILD_HASHTABLE->非并行执行,切换状态为HJ_NEED_NEW_OUTER

HJ_NEED_NEW_OUTER->获取(执行ExecHashJoinOuterGetTuple)下一个outer relation的一个元组

```
349 if (parallel)
(gdb) n
354 outerTupleSlot =
(gdb)
357 if (TupIsNull(outerTupleSlot))
(gdb) p *outerTupleSlot
$20 = {type = T_TupleTableSlot, tts_isempty = false, tts_shouldFree = false, tts_shouldFreeMin = false, tts_slow = true,
  tts_tuple = 0x2f88300, tts_tupleDescriptor = 0x7f0710d02bd0, tts_mcxt = 0x2ee1640, tts_buffer = 507, tts_nvalid = 1,
  tts_values = 0x2ee22a8, tts_isnull = 0x2ee22d0, tts_mintuple = 0x0, tts_minhdr = {t_len = 0, t_self = {ip_blkid = {
   bi_hi = 0, bi_lo = 0}, ip_posid = 0}, t_tableOid = 0, t_data = 0x0}, tts_off = 2, tts_fixedTupleDescriptor = true}
```

HJ_NEED_NEW_OUTER->切换状态为HJ_SCAN_BUCKET,开始扫描Hash Table

HJ_SCAN_BUCKET->不匹配,切换状态为

```
(gdb)
```

```
416 if (parallel)
(gdb) n
427 if (!ExecScanHashBucket(node, econtext))
(gdb)
430 node->hj_JoinState = HJ_FILL_OUTER_TUPLE;
(gdb)
431 continue;
(gdb)
```

HJ_FILL_OUTER_TUPLEHJ_FILL_OUTER_TUPLE->切换状态为HJ_NEED_NEW_OUTER

不管是否获得/发出一个元组，下一个状态是NEED_NEW_OUTER

HJ_SCAN_BUCKET->存在匹配的元组,设置相关标记

```
(gdb) n
449 node->hj_MatchedOuter = true;
(gdb)
450 HeapTupleHeaderSetMatch(HJTUPLE_MINTUPLE(node->hj_CurTuple));
(gdb)
453 if (node->js.jointype == JOIN_ANTI)
(gdb) n
464 if (node->js.single_match)
(gdb)
 465 node->hj_JoinState = HJ_NEED_NEW_OUTER;
 (gdb)
```

4.任务

4.1 根据2.2 定位Index Scan的节点并打印其内容

```
postgres=# explain select * from t1,t2 where t1.c1<5 and t1.c2 = t2.c2;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=8.41..228.81 rows=40 width=16)
   Hash Cond: (t2.c2 = t1.c2)
   ->  Seq Scan on t2  (cost=0.00..145.00 rows=10000 width=8)
   ->  Hash  (cost=8.36..8.36 rows=4 width=8)
         ->  Index Scan using t1_pkey on t1  (cost=0.29..8.36 rows=4 width=8)
               Index Cond: (c1 < 5)
(6 rows)
```

4.2 调试 HashJoin 执行过程中node->hj_JoinState 的变化过程，解释PG Hash执行过程

参考

1.https://www.pgcon.org/2017/schedule/events/1053.en.html

2.https://www.jianshu.com/p/016ab35c41fd