数据库引论
实验报告

实验 6　Block Nested Loop Join

学生姓名：　　彭恺欣

学　　号：　　21307140077

日　　期：　2024 年 6 月 14 日

一、 实验要求

准备Linux开发环境，进行必要的配置。

下载Postgres安装文件源码，进行编译安装。

理解Postgres的源码。

修改源码，并调试。

完成后重新编译，安装Postgres。

运行实验测试例子，记录运行花费时间。

撰写实验报告。

二、 实验过程

（一）、Postgresql对于Nest Loop Join指令的实现逻辑

(1) 解析阶段：用户输入与Join相关的SQL查询语句，系统接收并调用backend/parser/parser.c中的raw_parser函数对其进行初始的分析，生成原始查询树。

(2) 重写阶段：原始查询树输入重写系统进行规则分析和结构重写，将整合的查询转化为相互连接的单独节点，其中包括NestLoopJoin节点；这个过程调用backend/tcop/postgres.c中的pg_analyze_and_rewrite函数实现。

(3) 生成查询计划并优化：重写后的查询树进入规划系统进行规划和优化，首先根据查询树生成初始化的计划，然后分析最优的查询路径并生成最终的查询计划(Plan)，这一过程主要在文件backend/optimizer/plan/planner.c中实现。

(4) 执行阶段：根据查询计划，对计划节点进行执行。所有节点首先在backend/executor/execProcnode.c中进行初始化，然后NestLoopJoin节点具体在backend/executor/nodeNestloop.c中实现。

具体的，ExecInitNestLoop函数对节点进行针对性的初始化，包括节点状态、表达式、元组存储结构和执行时需要用到的标志等信

息；ExecNestLoop开始执行Join算法，首先遍历外表获取外部元组，然后遍历内表，直到函数返回一对匹配的元组构成的Join结果元组或者内表遍历结束，此时回到外表遍历下一个外部元组并重新扫描内表；ExecEndNestLoop函数负责在Join节点执行完毕后进行空间释放等清理工作；ExecReScanNestLoop函数负责在外表发生改变时自动重新扫描外表。

(5) 返回结果：在backend/executor/execMain.c通过重复上述执行阶段得到Join最终的结果，并继续查询计划的执行直到所有查询完成。

（二）、修改相关函数实现BlockNestLoopJoin

(1) backend/utils/misc/guc.c

include/postmaster/bg_worker.h

```
extern int block_nested_loop_join_block_size = 1;
static struct config_int ConfigureNamesInt[] ={
    {
        {"bnlj_block_size", PGC_USERSET, QUERY_TUNING_METHOD,
            gettext_noop("Sets the block size for Block Nested Loop Join."),
            NULL, GUC_UNIT_BLOCKS
        },
        &block_nested_loop_join_block_size,
        1, 1, 1024,
        NULL, NULL, NULL
    },
    /**************
     * Original codes
    **************/
}
```

通过指定全局变量block_nested_loop_join_block_size并在guc.c中的结构ConfigureNamesInt[]内添加以上代码，实现对算法外部元组块大小的动态调整，可以在运行时通过指令”SET blnj_block_size = K;”将接下来BlockNestLoopJoin算法支持的块大小设置为K(初始化K=1,且限定K<=1024)。

(2) backend/include/nodes/execnodes.h

```
typedef struct NestLoopState{
    /*****************************
     * Original codes for NestLoopJoin
     *****************************/

    /* Block Nested Loop Join*/
    TupleTableSlot **outerBlock;    // Block to restore OuterTupleTableSlots
    int blockSize;    // Size of the block
    int Bsize; // Number of TupleTableSlots in the current block
    bool terminal; // BNLJ terminated signal
} NestLoopState;
```

在NestLoopState中添加与BlockNestLoopJoin相关的定义，包括以下结构：外部元组块指针(TupleTableSlot **outerBlock)、块大小(int blockSize)、当前遍历中块存储的元组数量(int Bsize)以及算法结束信号(bool terminal)。

(3) backend/executor/nodeNestloop.c

1. ExecInitNestLoop函数

```
NestLoopState * ExecInitNestLoop(NestLoop *node, EState *estate, int eflags){
    /*****************************
     * Original codes for NestLoopJoin
     *****************************/
    /* initialize block nested loop join */
    nlstate->blockSize = block_nested_loop_join_block_size;
    nlstate->outerBlock = (TupleTableSlot **) palloc0(nlstate->blockSize *
sizeof(TupleTableSlot *));
    nlstate->Bsize = 0;
    nlstate->terminal = false;
    /* initialize the TupleTableSlots in the outer block */
    for (int i = 0; i < nlstate->blockSize; i++){
        nlstate->outerBlock[i] = ExecInitExtraTupleSlot(estate,
            ExecGetResultType(outerPlanState(nlstate)),
            ExecGetResultSlotOps(outerPlanState(nlstate), NULL));
    }
    return nlstate;
}
```

在对原本NestLoopJoin节点信息初始化的同时添加与

BlockNestLoopJoin相关信息的初始化，包括：将块大小(blockSize)初始化为用户设置的值，为外部元组块指针(blockSize)申请相应大小的空间，将最初块中元组数量设置为0，同时将终止信号设置为False；额外的，对元组块中的每个元组槽(TupleTableSlot)根据当前计划节点(nlstate)进行初始化。

2. ExecNestLoop函数

```
static TupleTableSlot * ExecNestLoop(PlanState *pstate){
    /**************************
     * Original codes for NestLoopJoin
     **************************/
    for (;;){
        /*
         * If we have an empty or out-of-time outer block, get a new block and reset
         * the inner scan.
         */
        if (node->nl_NeedNewOuter){
            node->Bsize = 0;
            for (int i = 0; i < node->blockSize; i++){
                outerTupleSlot = ExecProcNode(outerPlan);
                if (TupIsNull(outerTupleSlot)){
                    node->terminal = true;
                    break;
                }
                ExecCopySlot(node->outerBlock[i], outerTupleSlot);
                node->Bsize++;
            }
            if (node->Bsize == 0){
                ENL1_printf("no outer tuple, ending join");
                return NULL;
            }
            node->nl_NeedNewOuter = false;
            /*now rescan the inner plan*/
            ENL1_printf("rescanning inner plan");
            ExecReScan(innerPlan);
        }
```

从主循环开始，首先根据nl_NeedNewOuter标志判断当前块对应的内表循环是否完成，如果完成则获取新的外部元组填充当前块，

更新块中元组数量(Bsize)和nl_NeedNewOuter标志，并且重新扫描内表实现对内表的重新遍历。特别的，当块中元组数量为0时表示外表循环结束，此时算法结束；当块中元组数量(Bsize)小于块大小(blockSize)时，设置终止信号(terminal)说明当前块是最后的块。

```
/*we have an outer block, try to get the next inner tuple.*/
innerTupleSlot = ExecProcNode(innerPlan);
econtext->ecxt_innertuple = innerTupleSlot;
if (TupIsNull(innerTupleSlot)){
    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)){
        /*
         * 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;
    }
    else if (node->terminal){
        ENL1_printf("no inner tuple, ending join");
        return NULL;
    }
    /* Otherwise just return to top of loop to get the next inner tuple.*/
    continue;
}
```

对于块判断和更新操作完成后，我们开始对内表进行遍历，通过ExecProcNode(innerPlan)获取一个新的内表元组；当内表遍历结束时将nl_NeedNewOuter设置为真表示需要更新外部元组块，然后根据不同Join类型决定是否设置空元组，否则根据终止信息判断是否结束循环。

```
/* Get each outer tuple in the outer block and test the qualification.*/
for (int i = 0; i < node->Bsize; i++){
    econtext->ecxt_outertuple = node->outerBlock[i];
    node->nl_MatchedOuter = false;
    /*
     * Ftch the values of any outer Vars and store them in the
     * appropriate PARAM_EXEC slots.
     */
    foreach(lc, nl->nestParams){
        /**************************
         * Original codes for NestLoopJoin
         **************************/
    }
    /*
    * 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)){
        node->nl_MatchedOuter = true;
        /* In an antijoin, we never return a matched tuple */
        if (node->js.jointype == JOIN_ANTI)
            continue;
        /*
        * If we only need to join to the first matching inner tuple, then
        * consider returning this one, but after that continue with next
        * outer tuple.
        */
        if (node->js.single_match)
            break;
        if (otherqual == NULL || ExecQual(otherqual, econtext)){
            /*
            * qualification was satisfied so we project and return the
            * slot containing the result tuple using ExecProject().
            */
            ENL1_printf("qualification succeeded, projecting tuple");
            return ExecProject(node->js.ps.ps_ProjInfo);
        }else InstrCountFiltered2(node, 1);
    }else InstrCountFiltered1(node, 1);
}
```

针对获取的内表元组，遍历每一个块中的外部元组；针对获取的外部元组，设置执行参数并将其与内部元组配对，判断这对元组是否符合Join要求，如果满足要求则返回结果否则继续遍历。

3. ExecEndNestLoop函数

```
void ExecEndNestLoop(NestLoopState *node){
    NL1_printf("ExecEndNestLoop: %s\n", "ending node processing");
    /* Free the exprcontext*/
    ExecFreeExprContext(&node->js.ps);
    /* Clean out the tuple table*/
    ExecClearTuple(node->js.ps.ps_ResultTupleSlot);
    /* Release the memory for outer block TupleTableSlot*/
    if (node->outerBlock){
        for (int i = 0; i < node->blockSize; i++){
            if (node->outerBlock[i])
                ExecClearTuple(node->outerBlock[i]);
        }
    }
    /* Close down subplans*/
    ExecEndNode(outerPlanState(node));
    ExecEndNode(innerPlanState(node));
    NL1_printf("ExecEndNestLoop: %s\n", "node processing ended");
}
```

添加了释放外部元组块所占用空间的操作。

（其他细节参考代码源文件）

三、 实验分析

（一）、实验结果

针对以上对于 Block Nest Loop Join 的实现，我在 Similarity 数据上通过以下指令测试了不同块大小对于算法速度的影响。

```
\c similarity
\timing
set enable_mergejoin to off;
set enable_hashjoin to off;
set bnlj_block_size = $BLOCK_SIZE;
SELECT count(*) FROM restaurantaddress ra, restaurantphone rp WHERE ra.name =
rp.name;
```

具体表现如下表：

块大小 blockSize	运行时间(ms)	加速比
1	394.741	0.00%
2	292.853	25.81%
4	266.633	32.38%
8	232.949	40.98%
16	219.126	44.48%
32	207.420	47.45%
64	204.312	48.24%

（二）、结果分析

可以看到，随着块大小增加，同一个 Join 指令的运行时间逐渐减小，但减小的速率趋于平缓，当块大小大于 32 之后算法运行速度几乎不再随着块大小增加而加快。

根据 Block Nest Loop Join 的实现逻辑，我们可以做如下假设：

令外表元组数量为N，内表元组数量为M，块大小为B；令每次 I/O 操作获取元组耗时为t，而通过内存获取元组耗时忽略不计；假设内存大小不足以存储所有内表元组。那么 Block Nest Loop Join 算法运行时间可以估计为$\left(\frac{N}{B}+NM\right)t$，因为每个外部元组块在更时以及每次遍历到的内表元组时需要通过 I/O 获取，而块内的外部元组可以通过内存获取。因此可以预见，随着块大小B增大运行时间逐渐减小，但是当B达到一定大小之后，继续增大B对于运行时间的影响变得很小，最后运行时间将会趋于常数NMt。

四、 总结与思考

（1）当循环意外无法终止时，虽然源代码可以正常编译，但是在初始化数据库时会失败；因为这个问题卡了好久，最后设置了终止条

件之后改好了。

（2）在测试时需要将 MergeJoin 和 HashJoin 关闭，否则有时候优化器会自动选择这两种 Join 而不是我们实现的 Block Nest Loop Join。