# 1. openGauss算子概述

## 1.1 openGauss执行算子汇总

openGauss的算子按类型可分为四类：控制算子、连接算子、扫描算子和物化算子。下面汇总了当前（openGauss2.0.0）已有的算子。

|  |  |  |
| --- | --- | --- |
| 算子 | 文件 | 类型 |
| Agg | nodeAgg.cpp | 物化算子 |
| Append | nodeAppend.cpp | 控制算子 |
| BitmapAnd | nodeBitmapAnd.cpp | 控制算子 |
| BitmapHeapscan | nodeBitmapHeapscan.cpp | 扫描算子 |
| BitmapIndexscan | nodeBitmapIndexscan.cpp | 扫描算子 |
| BitmapOr | nodeBitmapOr.cpp | 控制算子 |
| Ctescan | nodeCtescan.cpp | 扫描算子 |
| Foreignscan | nodeForeignscan.cpp | 扫描算子 |
| Functionscan | nodeFunctionscan.cpp | 扫描算子 |
| Group | nodeGroup.cpp | 物化算子 |
| Hash | nodeHash.cpp | 物化算子 |
| Hashjoin | nodeHashjoin.cpp | 连接算子 |
| Indexonlyscan | nodeIndexonlyscan.cpp | 扫描算子 |
| Indexscan | nodeIndexscan.cpp | 扫描算子 |
| Limit | nodeLimit.cpp | 物化算子 |
| LockRows | nodeLockRows.cpp | 控制算子 |
| Material | nodeMaterial.cpp | 物化算子 |
| MergeAppend | nodeMergeAppend.cpp | 控制算子 |
| Mergejoin | nodeMergejoin.cpp | 连接算子 |
| ModifyTable | nodeModifyTable.cpp | 控制算子 |
| Nestloop | nodeNestloop.cpp | 连接算子 |
| PartIterator | nodePartIterator.cpp | 连接算子 |
| Recursiveunion | nodeRecursiveunion.cpp | 控制算子 |
| Result | nodeResult.cpp | 控制算子 |
| Samplescan | nodeSamplescan.cpp | 扫描算子 |
| Seqscan | nodeSeqscan.cpp | 扫描算子 |
| SetOp | nodeSetOp.cpp | 物化算子 |
| Sort | nodeSort.cpp | 物化算子 |
| Stub | nodeStub.cpp | 控制算子 |
| Subplan | nodeSubplan.cpp | 控制算子 |
| Subqueryscan | nodeSubqueryscan.cpp | 扫描算子 |
| Tidscan | nodeTidscan.cpp | 扫描算子 |
| Unique | nodeUnique.cpp | 物化算子 |
| Valuesscan | nodeValuesscan.cpp | 扫描算子 |
| WindowAgg | nodeWindowAgg.cpp | 物化算子 |
| Worktablescan | nodeWorktablescan.cpp | 扫描算子 |
| Extensible | nodeExtensible.cpp | 用于扩展算子 |

## 1.2 PG新增算子汇总

下面列出PG最新版本（14devel）多了哪些算子。

|  |  |  |
| --- | --- | --- |
| 算子 | 文件 | 类型 |
| Custom | nodeCustom.c |  |
| Gather | nodeGather.c |  |
| GatherMerge | nodeGatherMerge.c |  |
| IncrementalSort | nodeIncrementalSort.c |  |
| Namedtuplestorescan | nodeNamedtuplestorescan.c |  |
| ProjectSet | nodeProjectSet.c |  |
| TableFuncscan | nodeTableFuncscan.c |  |
| Tidrangescan | nodeTidrangescan.c |  |

# 2. 算子插件（TidRangeScan）

1.1表格中的算子Extensible类似于PG的算子Custom，其作用是允许插件向数据库增加新的算子类型。主要分为三步：首先，在路径规划期间生成插件增加的扫描路径（ExtensiblePath）；然后，如果优化器选择该路径作为最优路径，那么需要生成对应的计划（ExtensiblePlan）；最后，必须提供执行该计划的能力（ExtensiblePlanState）。

下面以TidRangeScan为示例，演示如何使用Extensible通过插件化的方式为openGauss新增一个执行算子。

## 2.1 功能介绍

openGauss中堆表由一个个page组成，每个page包含若干个tuple。tid是tuple的寻址地址，由两个字段组成：（pageid,itemid），pageid代表第几个数据块，itemid代表这个page内的第几条记录。例如tid=(10,1)表示第11个数据块中的第一条记录（pageid从0开始，itemid从1开始）。

PostgreSQL 14 devel新增了算子TidRangeScan，可以直接通过tid来范围访问某个page的全部数据。（带来的好处：如果需要更新一张表所有数据时，可以开启多个会话并行去更新不同的page，提高效率。）

本次展示将该特性通过插件的方式移植到openGauss，插件化的增加一个执行算子。

## 2.2 使用说明

tidrangescan插件定义了一个bool类型的guc参数：enable\_tidrangescan，控制优化器对tidrangescan扫描算子的使用，on表示使用，off表示不使用。



## 2.3 插件边界

本小节主要列举调用了哪些内核接口，当内核演进过程中修改了这些接口，有可能会影响插件的使用。

|  |  |
| --- | --- |
| 接口名 | 文件 |
| ExecInitExpr | execQual.cpp |
| clauselist\_selectivity | clausesel.cpp |
| cost\_qual\_eval | costsize.cpp |
| get\_tablespace\_page\_costs | spccache.cpp |
| get\_baserel\_parampathinfo | relnode.cpp |
| add\_path | pathnode.cpp |
| extract\_actual\_clauses | restrictinfo.cpp |
| heap\_getnext | heapam.cpp |
| ExecClearTuple | execTuples.cpp |
| ExecStoreTuple | execTuples.cpp |
| ExecScanReScan | execScan.cpp |
| heap\_beginscan | heapam.cpp |
| heap\_rescan | heapam.cpp |
| ExecScan | execScan.cpp |
| heap\_endscan | heapam.cpp |
| make\_ands\_explicit | clauses.cpp |
| deparse\_context\_for\_planstate | ruleutils.cpp |
| deparse\_expression | ruleutils.cpp |
| ExplainPropertyText | explain.cpp |

## 2.4 设计实现

本节提到的hook在第3章《hook点总述》会做详细说明。

### 2.4.1 优化器

#### 2.4.1.1 添加路径

将set\_rel\_pathlist\_hook赋值为SetTidRangeScanPath，该函数解析扫描表的条件，当存在tid范围查询时调用add\_path添加ExtensiblePath，计算代价，并将创建计划的接口tidrangescan\_path\_methods存入path中。

static void SetTidRangeScanPath(PlannerInfo \*root, RelOptInfo \*baserel,

                Index rtindex, RangeTblEntry \*rte)

{

 ...

    tidrangequals = TidRangeQualFromRestrictInfoList(baserel->baserestrictinfo, baserel);

 ...

    if (tidrangequals != NIL)

    {

        cpath = (ExtensiblePath\*)palloc0(sizeof(ExtensiblePath));

        cpath->path.type = T\_ExtensiblePath;

        cpath->path.pathtype = T\_ExtensiblePlan;

        cpath->path.parent = baserel;

        cpath->extensible\_private = tidrangequals;

        cpath->methods = &tidrangescan\_path\_methods;

        cost\_tidrangescan(&cpath->path, root, baserel, tidrangequals,

                      cpath->path.param\_info);

        add\_path(root, baserel, &cpath->path);

    }

}

static ExtensiblePathMethods    tidrangescan\_path\_methods = {

    "tidrangescan",                /\* ExtensibleName \*/

    PlanTidRangeScanPath,        /\* PlanExtensiblePath \*/

};

####  2.4.1.2 创建计划

上述的tidrangescan\_path\_methods定义了创建计划函数PlanTidRangeScanPath，根据最优路径生成计划ExtensiblePlan，同时将创建计划状态节点接口tidrangescan\_scan\_methods存入plan。

static Plan \* PlanTidRangeScanPath(PlannerInfo \*root,

                 RelOptInfo \*rel,

                 ExtensiblePath \*best\_path,

                 List \*tlist,

                 List \*clauses,

                 List \*custom\_plans)

{

    ExtensiblePlan \*node = makeNode(ExtensiblePlan);

    Plan       \*plan = &node->scan.plan;

    List       \*tidrangequals = best\_path->extensible\_private;

    ...

    node->extensible\_exprs = tidrangequals;

    node->scan.plan.startup\_cost = best\_path->path.startup\_cost;

    node->scan.plan.total\_cost = best\_path->path.total\_cost;

    node->scan.plan.plan\_rows = best\_path->path.rows;

    node->scan.plan.plan\_width = rel->width;

    node->methods = &tidrangescan\_scan\_methods;

    return plan;

}

static ExtensiblePlanMethods    tidrangescan\_scan\_methods = {

    "tidrangescan",                /\* ExtensibleName \*/

    CreateTidRangeScanState,    /\* CreateExtensiblePlanState \*/

};

###  2.4.2 执行器

#### 2.4.2.1 创建计划状态节点

上述的tidrangescan\_scan\_methods定义了创建PlanState函数CreateTidRangeScanState，根据传入的plan返回PlanState，同样将后续执行器执行的若干方法结构体tidrangescan\_exec\_methods存入PlanState。

Node\* CreateTidRangeScanState(ExtensiblePlan \*custom\_plan)

{

    TidRangeScanState \*tidrangestate;

    /\*

     \* create state structure

     \*/

    tidrangestate = (TidRangeScanState\*)palloc0(sizeof(TidRangeScanState));

    NodeSetTag(tidrangestate, T\_ExtensiblePlanState);

    tidrangestate->css.methods = &tidrangescan\_exec\_methods;

    /\*

     \* mark scan as not in progress, and TID range as not computed yet

     \*/

    tidrangestate->trss\_inScan = false;

    return (Node\*)&tidrangestate->css;

}

static ExtensibleExecMethods    tidrangescan\_exec\_methods = {

    "tidrangescan",                /\* ExtensibleName \*/

    BeginTidRangeScanScan,            /\* BeginExtensiblePlan \*/

    ExecTidRangeScan,            /\* ExecExtensiblePlan \*/

    EndTidRangeScan,            /\* EndExtensiblePlan \*/

    ExecReScanTidRangeScan,            /\* ReScanExtensiblePlan \*/

    ExplainTidRangeScan             /\* ExplainExtensiblePlan \*/

};

#### 2.4.2.2 执行层hook

tidrangescan\_exec\_methods定义了五个接口，分别是执行层各个阶段的主函数：BeginTidRangeScanScan、ExecTidRangeScan、EndTidRangeScan、ExecReScanTidRangeScan、ExplainTidRangeScan。

static void BeginTidRangeScanScan(ExtensiblePlanState \*node, EState \*estate, int eflags)

{

    TidRangeScanState  \*ctss = (TidRangeScanState \*) node;

    ExtensiblePlan       \*cscan = (ExtensiblePlan \*) node->ss.ps.plan;

    ctss->css.ss.ss\_currentScanDesc = NULL;    /\* no table scan here \*/

    /\*

     \* initialize child expressions

     \*/

    ctss->css.ss.ps.qual = (List\*)ExecInitExpr((Expr\*)cscan->scan.plan.qual, (PlanState \*)ctss);

    TidExprListCreate(ctss);

}

static TupleTableSlot \* ExecTidRangeScan(ExtensiblePlanState \*pstate)

{

    return ExecScan(&pstate->ss,

            (ExecScanAccessMtd) TidRangeNext,

            (ExecScanRecheckMtd) TidRangeRecheck);

}

static void EndTidRangeScan(ExtensiblePlanState \*node)

{

    TableScanDesc scan = node->ss.ss\_currentScanDesc;

    if (scan != NULL)

        heap\_endscan(scan);

    /\*

     \* Free the exprcontext

     \*/

    ExecFreeExprContext(&node->ss.ps);

    /\*

     \* clear out tuple table slots

     \*/

    if (node->ss.ps.ps\_ResultTupleSlot)

        ExecClearTuple(node->ss.ps.ps\_ResultTupleSlot);

    ExecClearTuple(node->ss.ss\_ScanTupleSlot);

}

static void ExecReScanTidRangeScan(ExtensiblePlanState \*node)

{

    /\* mark scan as not in progress, and tid range list as not computed yet \*/

    ((TidRangeScanState\*)node)->trss\_inScan = false;

    /\*

     \* We must wait until TidRangeNext before calling table\_rescan\_tidrange.

     \*/

    ExecScanReScan(&node->ss);

}

static void ExplainTidRangeScan(ExtensiblePlanState \*node, List \*ancestors, ExplainState \*es)

{

    TidRangeScanState  \*ctss = (TidRangeScanState \*) node;

    ExtensiblePlan       \*cscan = (ExtensiblePlan \*) ctss->css.ss.ps.plan;

    /\* logic copied from show\_qual and show\_expression \*/

    if (cscan->extensible\_exprs)

    {

        bool    useprefix = es->verbose;

        Node   \*qual;

        List   \*context;

        char   \*exprstr;

        /\* Convert AND list to explicit AND \*/

        qual = (Node \*) make\_ands\_explicit(cscan->extensible\_exprs);

        /\* Set up deparsing context \*/

        context = deparse\_context\_for\_planstate((Node\*)ctss, ancestors, es->rtable);

        /\* Deparse the expression \*/

        exprstr = deparse\_expression(qual, context, useprefix, false);

        /\* And add to es->str \*/

        ExplainPropertyText("tid range quals", exprstr, es);

    }

}

# 3. hook点总述

## 3.1 优化器

### 3.1.1 添加路径

通常用来产生ExtensiblePath对象，并使用add\_path把它们加入到rel中。

插入位置所在的函数：set\_rel\_pathlist

typedef void (\*set\_rel\_pathlist\_hook\_type) (PlannerInfo \*root,

                                           RelOptInfo \*rel,

                                           Index rti,

                                           RangeTblEntry \*rte);

extern THR\_LOCAL PGDLLIMPORT set\_rel\_pathlist\_hook\_type set\_re

l\_pathlist\_hook;

ExtensiblePath定义如下。

typedef struct ExtensiblePath {

    Path path;

    uint32 flags;           /\* mask of EXTENSIBLEPATH\_\* flags \*/

    List\* extensible\_paths; /\* list of child Path nodes, if any \*/

    List\* extensible\_private;

    const struct ExtensiblePathMethods\* methods;

} ExtensiblePath;

\* flags是一个标识，如果该自定义的路径支持反向扫描，则它应该包括EXTENSIBLEPATH\_SUPPORT\_BACKWARD\_SCAN，如果支持标记和恢复则包括EXTENSIBLEPATH\_SUPPORT\_MARK\_RESTORE。

\* extensible\_paths是这个自定义路径节点的子Path节点列表

\* extensible\_private可用来存储该自定义路径的私有数据。

\* methods必须包含根据该路径生成计划的方法。ExtensiblePathMethods结构如下，主要实现PlanExtensiblePath。

typedef struct ExtensiblePathMethods {

    const char\* ExtensibleName;

    /\* Convert Path to a Plan \*/

struct Plan\* (\*PlanExtensiblePath)(PlannerInfo\* root, RelOptInfo\* rel, struct ExtensiblePath\* best\_path,

 List\* tlist, List\* clauses, List\* extensible\_plans);

} ExtensiblePathMethods;

### 3.1.2 添加连接路径

提供连接路径，同样创建ExtensiblePath路径。

插入位置所在的函数：add\_paths\_to\_joinrel

typedef void (\*set\_join\_pathlist\_hook\_type) (PlannerInfo \*root,

                                               RelOptInfo \*joinrel,

                                               RelOptInfo \*outerrel,

                                               RelOptInfo \*innerrel,

                                               JoinType jointype,

                                               SpecialJoinInfo \*sjinfo,

                                               Relids param\_source\_rels,

                                               SemiAntiJoinFactors \*semifactors,

                                               List \*restrictlist);

extern THR\_LOCAL PGDLLIMPORT set\_join\_pathlist\_hook\_type set\_join\_pathlist\_hook;

### 3.1.3 创建计划

调用上述ExtensiblePath中的methods定义的接口PlanExtensiblePath，将自定义路径转换为一个完整的计划，返回ExtensiblePlan。

插入位置所在的函数：create\_scan\_plan->create\_extensible\_plan

typedef struct ExtensiblePlan {

    Scan scan;

    uint32 flags;                  /\* mask of EXTENSIBLEPATH\_\* flags, see relation.h \*/

    List\* extensible\_plans;        /\* list of Plan nodes, if any \*/

    List\* extensible\_exprs;        /\* expressions that extensible code may evaluate \*/

    List\* extensible\_private;      /\* private data for extensible code \*/

    List\* extensible\_plan\_tlist;   /\* optional tlist describing scan

                                    \* tuple \*/

    Bitmapset\* extensible\_relids;  /\* RTIs generated by this scan \*/

    ExtensiblePlanMethods\* methods;

} ExtensiblePlan;

\* 和ExtensiblePath一样，flags同样是一个标识。

\* extensible\_plans可以用来存放子Plan节点

\* extensible\_exprs用来存储需要由setrefs.cpp和subselect.cpp修整的表达式树。

\* extensible\_private用来存储只有该自定义算子使用的私有数据。

\* extensible\_plan\_tlist描述目标列

\* extensible\_relids为该扫描节点要处理的关系集合

\* methods必须包含生成该计划对应的计划节点PlanState的方法。ExtensiblePlanMethods结构如下，主要实现CreateExtensiblePlanState。

typedef struct ExtensiblePlanMethods {

    char\* ExtensibleName;

    /\* Create execution state (ExtensiblePlanState) from a ExtensiblePlan plan node \*/

    Node\* (\*CreateExtensiblePlanState)(struct ExtensiblePlan\* cscan);

} ExtensiblePlanMethods;

## 3.2 执行器

### 3.2.1 创建计划状态节点

调用上述ExtensiblePlanMethods中的methods定义的接口CreateExtensiblePlanState,为这个ExtensiblePlan分配一个ExtensiblePlanState。

插入位置所在的函数：ExecInitNodeByType->ExecInitExtensiblePlan

typedef struct ExtensiblePlanState {

    ScanState ss;

    uint32 flags;        /\* mask of EXTENSIBLEPATH\_\* flags, see relation.h \*/

    List\* extensible\_ps; /\* list of child PlanState nodes, if any \*/

    const ExtensibleExecMethods\* methods;

} ExtensiblePlanState;

\* flags含义同ExtensiblePath和ExtensiblePlan一样

\* extensible\_ps为该计划节点的子节点。

\* methods为包含多个执行所需接口的结构体ExtensibleExecMethods，在下文做具体介绍。

### 3.2.2 执行层hook

上面CustomScanState的成员CustomExecMethods定义了几个hook点

typedef struct ExtensibleExecMethods {

    const char\* ExtensibleName;

    /\* Executor methods: mark/restore are optional, the rest are required \*/

    void (\*BeginExtensiblePlan)(struct ExtensiblePlanState\* node, EState\* estate, int eflags);

    TupleTableSlot\* (\*ExecExtensiblePlan)(struct ExtensiblePlanState\* node);

    void (\*EndExtensiblePlan)(struct ExtensiblePlanState\* node);

    void (\*ReScanExtensiblePlan)(struct ExtensiblePlanState\* node);

    void (\*ExplainExtensiblePlan)(struct ExtensiblePlanState\* node, List\* ancestors, struct ExplainState\* es);

} ExtensibleExecMethods;

1) BeginExtensiblePlan完成所提供的`ExtensiblePlanState`的初始化。标准的域已经被`ExecInitExtensiblePlan`初始化，但是任何私有的域应该在这里被初始化。

插入位置所在的函数：ExecInitNodeByType->ExecInitExtensiblePlan

2) ExecExtensiblePlan执行扫描，取下一个扫描元组，如果还有任何元组剩余，它应该用当前扫描方向的下一个元组填充`ps\_ResultTupleSlot`，并且接着返回该元组槽。如果没有，则用`NULL`填充或者返回一个空槽。

插入位置所在的函数：ExecProcNode->ExecProcNodeByType

3) EndExtensiblePlan清除任何与`ExtensiblePlanState`相关的私有数据。这个方法是必需的，但是如果没有相关的数据或者相关数据将被自动清除，则它不需要做任何事情。

插入位置所在的函数：ExecEndNodeByType->ExecEndExtensiblePlan

4) ReScanExtensiblePlan把当前扫描倒回到开始处，并且准备重新扫描该关系。

插入位置所在的函数：ExecReScan->ExecReScanByType

5) ExplainExtensiblePlan为一个自定义扫描计划节点的`EXPLAIN`输出额外的信息。这个回调函数是可选的。即使没有这个回调函数，被存储在`ScanState`中的公共的数据（例如目标列表和扫描关系）也将被显示，但是该回调函数允许显示额外的信息（例如私有状态）。

插入位置所在的函数：ExplainNode->show\_pushdown\_qual