universität innsbruck

# Architektur und Implementierung von Datenbanksystemen

## Task 6 – recursive query execution in PostgreSQL

Team 3 - Gründlinger Diana, Huber Marcel, Klotz Thomas, Targa Aaron, Thalmann Matthias

# Content

- Syntax of a recursive query
- Execution plan
- Operators used and execution control flow

# Syntax of a recursive query

```
WITH RECURSIVE  RecRel (attr1, attr2, ..., attrn)
    AS     (SFW-Statement1                    -- Rekursionsinitialisierung
            UNION ALL
            SFW-Statement2                    -- Rekursionsschritt
            )

SELECT  [ DISTINCT ] attributliste
    FROM  RecRel
    [ WHERE   ...] [ GROUP BY ... ] [ HAVING ... ] [ ORDER BY ... ]
```

(2)

# Syntax of a recursive query

- WITH → common table expression (CTE), basically defining a temporary table that just exists for one query
- Optional RECURSIVE modifier → WITH query can refer to its own output

General Form of a recursive WITH query:

- A non-recursive term
- UNION or UNION ALL
- A recursive term

```
WITH RECURSIVE  RecRel (attr1, attr2, ..., attrn)
    AS     (SFW-Statement1              -- Rekursionsinitialisierung
            UNION ALL
            SFW-Statement2              -- Rekursionsschritt
            )
SELECT  [ DISTINCT ] attributliste
    FROM  RecRel
    [ WHERE   ...] [ GROUP BY ... ] [ HAVING ... ] [ ORDER BY ... ]
```

Only the recursive term can reference the query's own output.

(1,2)

# Execution plan - example

```
Query:
WITH RECURSIVE recursive_query(id, child_id) AS (
    SELECT id, child_id
    FROM my_tree
    WHERE id = 0
  UNION
    SELECT t.id, t.child_id
    FROM my_tree t, recursive_query
    WHERE recursive_query.child_id = t.id
)
SELECT * FROM recursive_query
```

```
Explain output:
CTE Scan on recursive_query  (cost=113.31..115.33 rows=101 width=8) (actual time=0.686..1257.405 rows=500 loops=1)
   CTE recursive_query
     ->  Recursive Union  (cost=0.27..113.31 rows=101 width=8) (actual time=0.680..1254.706 rows=500 loops=1)
           ->  Index Scan using idx on my_tree  (cost=0.27..8.29 rows=1 width=8) (actual time=0.664..0.670 rows=1 loops=1)
                 Index Cond: (id = 0)
           ->  Hash Join  (cost=0.33..10.30 rows=10 width=8) (actual time=1.301..2.487 rows=1 loops=500)
                 Hash Cond: (t.id = recursive_query_1.child_id)
                 ->  Seq Scan on my_tree t  (cost=0.00..8.00 rows=500 width=8) (actual time=0.009..1.225 rows=500 loops=500)
                 ->  Hash  (cost=0.20..0.20 rows=10 width=4) (actual time=0.013..0.013 rows=1 loops=500)
                       Buckets: 1024  Batches: 1  Memory Usage: 9kB
                       ->  WorkTable Scan on recursive_query recursive_query_1  (cost=0.00..0.20 rows=10 width=4) (actual
time=0.003..0.005 rows=1 loops=500)
 Planning time: 0.688 ms
 Execution time: 1259.438 ms
```

(3)

# Execution plan - example

```
Query:
WITH RECURSIVE recursive_query(id, child_id) AS (
    SELECT id, child_id
    FROM my_tree
    WHERE id = 0
  UNION
    SELECT t.id, t.child_id
    FROM my_tree t, recursive_query
    WHERE recursive_query.child_id = t.id
)
SELECT * FROM recursive_query
```

```
Explain output:
CTE Scan on recursive_query  (cost=113.31..115.33 rows=101 width=8) (actual time=0.686..1257.405 rows=500 loops=1)
  CTE recursive_query
    ->  Recursive Union  (cost=0.27..113.31 rows=101 width=8) (actual time=0.680..1254.706 rows=500 loops=1)
          ->  Index Scan using idx on my_tree  (cost=0.27..8.29 rows=1 width=8) (actual time=0.664..0.670 rows=1 loops=1)
                Index Cond: (id = 0)
          ->  Hash Join  (cost=0.33..10.30 rows=10 width=8) (actual time=1.301..2.487 rows=1 loops=500)
                Hash Cond: (t.id = recursive_query_1.child_id)
                ->  Seq Scan on my_tree t  (cost=0.00..8.00 rows=500 width=8) (actual time=0.009..1.225 rows=500 loops=500)
                ->  Hash  (cost=0.20..0.20 rows=10 width=4) (actual time=0.013..0.013 rows=1 loops=500)
                      Buckets: 1024  Batches: 1  Memory Usage: 9kB
                      ->  WorkTable Scan on recursive_query recursive_query_1  (cost=0.00..0.20 rows=10 width=4) (actual
time=0.003..0.005 rows=1 loops=500)
 Planning time: 0.688 ms
 Execution time: 1259.438 ms
```

(3)

# Execution plan - example

```
Query:
WITH RECURSIVE recursive_query(id, child_id) AS (
    SELECT id, child_id
    FROM my_tree
    WHERE id = 0
  UNION
    SELECT t.id, t.child_id
    FROM my_tree t, recursive_query
    WHERE recursive_query.child_id = t.id
)
SELECT * FROM recursive_query
```

```
Explain output:
CTE Scan on recursive_query  (cost=113.31..115.33 rows=101 width=8) (actual time=0.686..1257.405 rows=500 loops=1)
  CTE recursive_query
    ->  Recursive Union  (cost=0.27..113.31 rows=101 width=8) (actual time=0.680..1254.706 rows=500 loops=1)
          ->  Index Scan using idx on my_tree  (cost=0.27..8.29 rows=1 width=8) (actual time=0.664..0.670 rows=1 loops=1)
                Index Cond: (id = 0)
          ->  Hash Join  (cost=0.33..10.30 rows=10 width=8) (actual time=1.301..2.487 rows=1 loops=500)
                Hash Cond: (t.id = recursive_query_1.child_id)
                ->  Seq Scan on my_tree t  (cost=0.00..8.00 rows=500 width=8) (actual time=0.009..1.225 rows=500 loops=500)
                ->  Hash  (cost=0.20..0.20 rows=10 width=4) (actual time=0.013..0.013 rows=1 loops=500)
                      Buckets: 1024  Batches: 1  Memory Usage: 9kB
                      ->  WorkTable Scan on recursive_query recursive_query_1  (cost=0.00..0.20 rows=10 width=4) (actual
time=0.003..0.005 rows=1 loops=500)
 Planning time: 0.688 ms
 Execution time: 1259.438 ms
```

(3)

# Execution plan - example

```
Query:
WITH RECURSIVE recursive_query(id, child_id) AS (
    SELECT id, child_id
    FROM my_tree
    WHERE id = 0
  UNION
    SELECT t.id, t.child_id
    FROM my_tree t, recursive_query
    WHERE recursive_query.child_id = t.id
)
SELECT * FROM recursive_query
```

```
Explain output:
CTE Scan on recursive_query  (cost=113.31..115.33 rows=101 width=8) (actual time=0.686..1257.405 rows=500 loops=1)
  CTE recursive_query
    ->  Recursive Union  (cost=0.27..113.31 rows=101 width=8) (actual time=0.680..1254.706 rows=500 loops=1)
          ->  Index Scan using idx on my_tree  (cost=0.27..8.29 rows=1 width=8) (actual time=0.664..0.670 rows=1 loops=1)
                Index Cond: (id = 0)
          ->  Hash Join  (cost=0.33..10.30 rows=10 width=8) (actual time=1.301..2.487 rows=1 loops=500)
                Hash Cond: (t.id = recursive_query_1.child_id)
                ->  Seq Scan on my_tree t  (cost=0.00..8.00 rows=500 width=8) (actual time=0.009..1.225 rows=500 loops=500)
                ->  Hash  (cost=0.20..0.20 rows=10 width=4) (actual time=0.013..0.013 rows=1 loops=500)
                      Buckets: 1024  Batches: 1  Memory Usage: 9kB
                      ->  WorkTable Scan on recursive_query recursive_query_1  (cost=0.00..0.20 rows=10 width=4) (actual
time=0.003..0.005 rows=1 loops=500)
 Planning time: 0.688 ms
 Execution time: 1259.438 ms
```

(3)

# Operators used

- CTE Scan
- Recursive Union
- WorkTable Scan

# Operators used and execution control flow

- **CTE Scan** – Scans results of a CTE as temporary table
- **WorkTable Scan** - Scans the work table used in evaluating a recursive CTE
- **Recursive Union** - Returns the union of the recursive and non recursive subplan

(4)

# Operators used and execution control flow

1. evaluate non-recursive term
   for UNION discard duplicate rows
   include remaining rows in result of recursive query, and also place them in a temporary working table

2. while working table not empty, repeat:
   a) Evaluate recursive term, substituting current contents of working table for recursive self-reference.
      For UNION (but not UNION ALL), discard duplicate rows and rows that duplicate any previous result row.
      Include all remaining rows in result of the recursive query, and also place them in temporary intermediate table.
   b) Replace contents of working table with contents of intermediate table, then empty intermediate table.

Execution terminates when no further tuples are added in a recursive step.
Note: It's called RECURSIVE but the process is really iterative.

(5)

```c
/* ----------------------------------------------------------------
 *      ExecRecursiveUnion(node)
 *
 *      Scans the recursive query sequentially and returns the next
 *      qualifying tuple.
 *
 * 1. evaluate non recursive term and assign the result to RT
 *
 * 2. execute recursive terms
 *
 * 2.1 WT := RT
 * 2.2 while WT is not empty repeat 2.3 to 2.6. if WT is empty returns RT
 * 2.3 replace the name of recursive term with WT
 * 2.4 evaluate the recursive term and store into WT
 * 2.5 append WT to RT
 * 2.6 go back to 2.2
 * ----------------------------------------------------------------
 */
static TupleTableSlot *
ExecRecursiveUnion(PlanState *pstate)
{
    RecursiveUnionState *node = castNode(RecursiveUnionState, pstate);
    PlanState  *outerPlan = outerPlanState(node);
    PlanState  *innerPlan = innerPlanState(node);
    RecursiveUnion *plan = (RecursiveUnion *) node->ps.plan;
    TupleTableSlot *slot;
    bool        isnew;

    CHECK_FOR_INTERRUPTS();

    /* 1. Evaluate non-recursive term */
    if (!node->recursing)
    {
        for (;;)
        {
            slot = ExecProcNode(outerPlan);
            if (TupIsNull(slot))
                break;
            if (plan->numCols > 0)
            {
                /* Find or build hashtable entry for this tuple's group */
                LookupTupleHashEntry(node->hashtable, slot, &isnew, NULL);
                /* Must reset temp context after each hashtable lookup */
                MemoryContextReset(node->tempContext);
                /* Ignore tuple if already seen */
                if (!isnew)
                    continue;
            }
            /* Each non-duplicate tuple goes to the working table ... */
            tuplestore_puttupleslot(node->working_table, slot);
            /* ... and to the caller */
            return slot;
        }
        node->recursing = true;
    }

    /* 2. Execute recursive term */
    for (;;)
    {
        slot = ExecProcNode(innerPlan);
        if (TupIsNull(slot))
        {
            /* Done if there's nothing in the intermediate table */
            if (node->intermediate_empty)
                break;

            /* done with old working table ... */
            tuplestore_end(node->working_table);

            /* intermediate table becomes working table */
            node->working_table = node->intermediate_table;

            /* create new empty intermediate table */
            node->intermediate_table = tuplestore_begin_heap(false, false,
                                                             work_mem);
            node->intermediate_empty = true;

            /* reset the recursive term */
            innerPlan->chgParam = bms_add_member(innerPlan->chgParam,
                                                 plan->wtParam);

            /* and continue fetching from recursive term */
            continue;
        }

        if (plan->numCols > 0)
        {
            /* Find or build hashtable entry for this tuple's group */
            LookupTupleHashEntry(node->hashtable, slot, &isnew, NULL);
            /* Must reset temp context after each hashtable lookup */
            MemoryContextReset(node->tempContext);
            /* Ignore tuple if already seen */
            if (!isnew)
                continue;
        }

        /* Else, tuple is good; stash it in intermediate table ... */
        node->intermediate_empty = false;
        tuplestore_puttupleslot(node->intermediate_table, slot);
        /* ... and return it */
        return slot;
    }

    return NULL;
}
```

(5)

# References


(1) https://www.postgresql.org/docs/current/queries-with.html
(2) Lecture slides from course “Database Systems”
(3) https://medium.com/swlh/postgres-recursive-query-cte-or-recursive-function-3ea1ea22c57c
(4) https://pganalyze.com/docs/explain#postgres-plan-nodes
(5) https://github.com/postgres/postgres/tree/master/src/backend/executor
(6) https://gitlab.com/postgres/postgres/blob/master/src/include/nodes/plannodes.h


www.uibk.ac.at