Eberhard Karls Universität Tübingen

Mathematisch-Naturwissenschaftliche Fakultät

Wilhelm-Schickard-Institut für Informatik

Masterthesis

Optimization of PL/pgSQL Translations Using Batching and Multiple Recursive References

Marcus Huber

Examiner

Prof. Dr. Torsten Grust

Co-Examiner

Prof. Dr. Klaus Ostermann

Supervisor

Denis Hirn

July 15, 2022

Abstract

Functions are rarely written to be executed only once. In the context of database applications, it is therefore often obvious to bundle function inputs and to calculate the query for all rows of a table at the same time. This so-called *batching* was used in the context of PL/pgSQL translations with recursive CTEs and refined under the use of multiple recursive references to a promising optimization method. The methods used were thereby explained and validated using various UDFs. We could observe an additional speedup by a factor of 2 and more compared to the translation, which reveals another clear advantage over PL/pgSQL. This speedup results here mainly from the use of `Hash Join`s instead of `Nested Loop Join`s. Furthermore, batching was also considered in the scope of a DBMS with *vectorization* like DuckDB where the performance gain can be extended to factors well above 1 000.

Contents

Introduction 1

Time is money. This is especially true nowadays when computing power is being rented or customers are becoming increasingly impatient and prefer to visit another website if there is even the slightest delay in loading time. Therefore, runtime optimization is one of the most important tools to be successful in the long run. We focus in this paper on further optimization of internal database queries by *batching* and the use of *multiple recursive references*. The starting point for us is the work of Denis Hirn and Torsten Grust [HG21], who have developed a compiler that translates PL/pgSQL functions into pure recursive SQL queries to avoid context switches. We now want to further optimize these translated recursive queries here concerning their runtime. Batching will be used to make better use of the tabular structure of relational database systems and multiple recursive references will be used to simplify the control flow within a function.
This work, therefore, introduces in Chapter 2.1 the translation of PL/pgSQL into recursive queries. We then explain the ideas of batching and multiple recursive references in chapters 2.2 and 2.3, before turning to migration to a current PostgreSQL version in Chapter 2.4. Chapter 3 then looks at these methods from the perspective of a second DBMS, namely DuckDB, which uses *vectorization*. Afterwards, Chapters 4.1 and 4.2 present the results using some UDFs (short for *User Defined Function*) and describe the limitations of our approaches.

Related Work

Optimizations in the context of database systems have been around for as long as there exist database systems. In this context, the topic of *batching*, as we now want to perform ourselves in the context of recursive queries, also arose early on and is still of great relevance today (cf. e.g. [MSD93, DKG18, Leo20]). The idea of multiple recursive references is also not new. For example, Microsoft SQL Server[1] already supports both multiple recursion starts and multiple recursive members. Similarly, IBM's DB2 supports multiple recursive references[2].
Our work is based on the translation method of [HG21], adapting these translations to a more recent PostgreSQL version. [HG21] in turn builds among others on *Froid* [RPE+17, RP19], which is an optimization of simpler non-looping imperative programs.

Short Overview of Recursive CTEs in PostgreSQL

Since all work in the context of PostgreSQL deals with recursive queries, we want to briefly repeat them here to provide an overview. This also prevents ambiguities in the nomenclature.

[1] https://docs.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql?view=sql-server-2017#guidelines-for-defining-and-using-recursive-common-table-expressions
[2] https://www.ibm.com/docs/en/i/7.4?topic=statement-using-recursive-queries

The following information and the minimal example used are taken from the PostgreSQL documentation[3].

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
      UNION ALL
    SELECT n+1
    FROM   t
    WHERE  n < 100)
SELECT sum(n)
FROM   t;
```

Listing 1.1: Minimal working example of a recursive CTE in PostgreSQL: Summation of all integers from 1 through 100.

You can see the general syntax from this example, which consists of a non-recursive term (also called *recursion start, recursion anchor, recursion initialization*) followed by a `UNION` or `UNION ALL` and a recursive term (also called *recursive member*), which can have a reference to itself. However, at most one self-reference may occur here, unless we use a trick or patch, which will be introduced later. Contrary to the naming `WITH RECURSIVE`, it is in fact an *iteration*. That is, the evaluation takes place using a temporary *working table* as follows:

- First, the start of the recursion is evaluated and written to the working table. As usual, duplicates are kept with `UNION ALL` and discarded using `UNION` only. In our example, we only get the value 1.
- Then the evaluation of the recursive part begins. New rows evaluated in the previous iteration are always accessed by the self-reference. That means we execute the iteration exactly as long as we create new rows. In our example, we have at the beginning the value 1 in the working table and pass the value 2 to the next iteration. This is successively repeated until we get the value 99 and pass 100. In the last iteration, no new row is created due to the filter and the iteration is aborted. The result consisting of the numbers 1 to 100 is then passed as a table `t` and summed up by the aggregate function.

In this work, we aim to find optimizations in both the recursion start and the recursive term by grouping function inputs and bypassing the limitation of having only one self-reference.

[3] https://www.postgresql.org/docs/14/queries-with.html#QUERIES-WITH-RECURSIVE

Background and Methodology 2

We want to build on the work of [HG21] and find new techniques to further improve the runtime of queries in PostgreSQL. As a running example, we have chosen the function *bbox* as defined in Listing 2.1, on which we can show in detail all the techniques presented below. This and most of the explored examples are taken from the GitHub repository mentioned in the paper.[1] *Bbox* detects the bounding box of a 2D object using the marching squares algorithm. For this purpose, we define a height map as a table using the type `vec2`, which is a two-dimensional vector (x, y). From this map, using a threshold, we generate the table `squares`, which stores the 2×2 vicinity for each pixel. The behaviour of the marching squares algorithm is stored in table `directions`. Based on the 2×2 vicinity, which direction to march is specified in the form of four boolean values `lower left (ll)`, `lower right (lr)`, `upper left (ul)` and `upper right (ur)`. The `track?` boolean is true if and only if one of the four pixels in the vicinity is set to true. This means `track?` indicates whether we have the object we are looking for in the immediate vicinity or whether we are further away from it. Then the algorithm starts at the `start` vector, looking for the vicinity `square` in the table `squares` and the direction `dir` the marching squares algorithm chooses to proceed. Once we have found the object, we go counter-clockwise to trace the shape and compute the bounding box out of it.

In the next sections, we will briefly recap the techniques used by [HG21] to translate PL/pgSQL UDFs in plain SQL queries using *recursive CTEs*. We will then introduce the idea of batching and using multiple recursive references to optimize query planning and develop a heuristic to find a nearly optimal new compilation strategy. At the end of the chapter, we will discuss the changes that we need to do when migrating these ideas from PostgreSQL 11.3 to the newer version 14.1 that we used here.

[1] http://github.com/One-WITH-RECURSIVE-is-Worth-Many-GOTOs

```
CREATE FUNCTION bbox(start vec2)
RETURNS box AS $$
  DECLARE
    "track?" boolean := false;
    goal     vec2;
    bbox     box := NULL;
    current  vec2 := start;
    square   squares;
    dir      directions;
  BEGIN
    WHILE true LOOP
      IF "track?" AND current = goal THEN
        EXIT;
      END IF;
      square := (SELECT s                                          -- Q1
                 FROM   squares AS s
                 WHERE  s.xy = current);
      dir    := (SELECT d                                          -- Q2
                 FROM   directions AS d
                 WHERE  (square.ll, square.lr, square.ul, square.ur) = (d.ll, d.lr,
                        d.ul, d.ur));
      IF NOT "track?" AND dir."track?" THEN
        "track?" := true;
        goal     := current;
        bbox     := box(point(goal.x, goal.y));
      END IF;
      IF "track?" THEN
        bbox := bound_box(bbox, box(point(current.x, current.y)));
      END IF;
      current := (current.x + (dir.dir).x, current.y + (dir.dir).y);
    END LOOP;
    RETURN bbox;
  END;
$$ LANGUAGE PLPGSQL STRICT;
```

Listing 2.1: Original PL/pgSQL UDF *bbox* with two embedded SQL queries Q_1 and Q_2.

2.1 Translation of PL/pgSQL

To avoid context switches due to embedded SQL queries, we use the compiler described in [HG21]. More precisely, we mostly use as a basis, if available, the already created translations, as they can be found in the GitHub repository. They transform the UDF into a *GOTO* style program in *single static assignment* form (SSA), perform a conversion into *administrative normal form* (ANF) and use *trampolined style* to fit into *WITH RECURSIVE* pattern of SQL. An intermediate representation of our example *bbox* as a control flow graph (CFG) is shown in Figure 2.1. The final plain SQL query with some manual optimizations is shown in Listing 2.2. These equivalent plain SQL queries can improve execution time by a factor of about 2, since context switches are time consuming as Denis Hirn and Torsten Grust have shown. We will not discuss the conversion steps further, but already use the translations as a starting point for our work. Note that we use `LATERAL` joins instead of `LATERAL LEFT OUTER` joins, since this is part of the migration to a newer version of PostgreSQL described in Section 2.4.

Each translation is based on the working table `run`, whose first column `rec?` indicates whether a column corresponds to an intermediate result or to the end of the calculation. The second column then always contains the result of the function call. The other columns correspond to the variables used by the UDF. We want as few columns as possible to minimize the space and

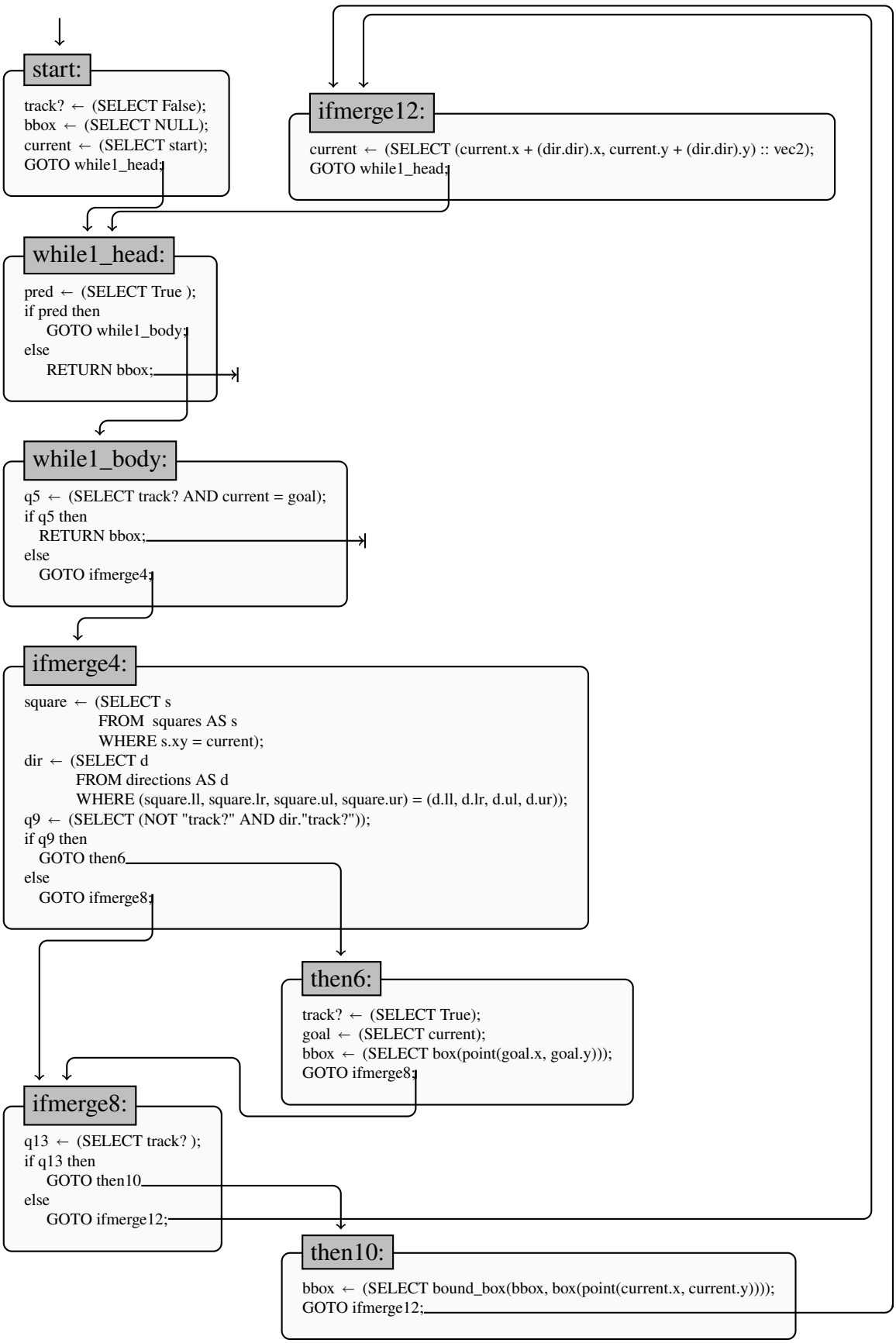


Figure 2.1: CFG for UDF *bbox* based on the CFG created by apfel-db compiler (https://apfel-db.informatik.uni-tuebingen.de/).

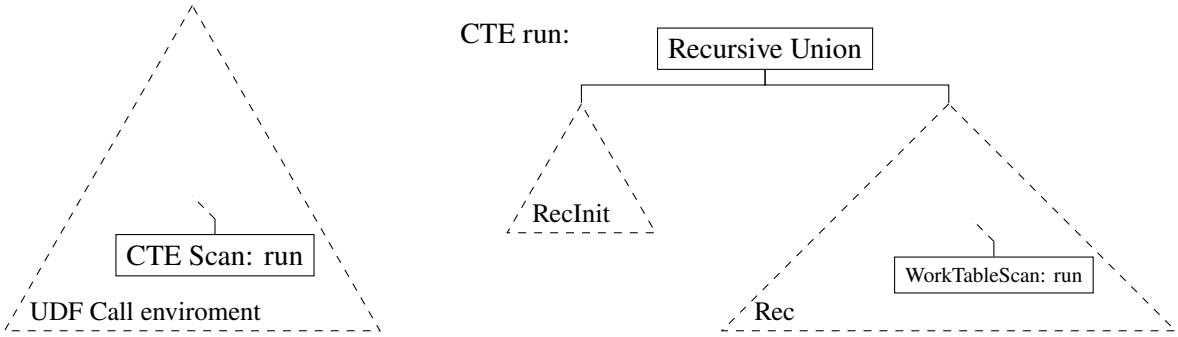


Figure 2.2: Sketch of a general SQL plan for translated UDFs using `WITH RECURSIVE`.

copy requirements inside the query. Therefore, for example, `square` and `dir` are not columns of `run`, since their values are overwritten in each iteration before referencing. In general, an additional column called `label` indicates which loop we are currently in. However, since this example can be inlined into one single loop, the label is constant `while1_head` and thus can be omitted as well. A final possible optimization would be to remove the `bbox` column by storing the variable `bbox` in `res` instead, rather than overwriting it at the end of the recursion. Now let's look at how CFG and translation are related. The start of recursion RA corresponds to the initial declaration inside `DECLARE` or equivalently to the start node inside the CFG. The recursive part A_0 corresponds to the remainder of the UDF or CFG. This is just the single while-loop represented by `while1_head`. Because the condition of the while-loop is always true, the return statement inside the head is unreachable code and is not included in the translation. The return statement in `while1_body` is represented by A_{0_0}. A_{0_1}, on the other hand, implements all paths that are still possible after entering `ifmerge4`. These are the following:

$A_{0_{1_0}}$: `then6` $\longrightarrow$ `ifmerge8` $\longrightarrow$ `then10` $\longrightarrow$ `ifmerge12` $\longrightarrow$ `while1_head`,

$A_{0_{1_1}}$: `ifmerge8` $\longrightarrow$ `then10` $\longrightarrow$ `ifmerge12` $\longrightarrow$ `while1_head` and

$A_{0_{1_2}}$: `ifmerge8` $\longrightarrow$ `ifmerge12` $\longrightarrow$ `while1_head`.

The remaining path via `then6`, `ifmerge8` and `ifmerge12` is not possible because `track` is set to true in `then6`. The variable assignments necessary to walk such a path can be found as filters in the WHERE clause of the individual blocks. E.g. A_{0_1} contains all paths that go through `ifmerge4` and therefore has the filter `(NOT (track? and current = goal))`. Now we have two possibilities to further optimize our UDF. If we assume that the UDF is to be executed more than once, we can investigate how to start these calls in a certain way at the same time, i.e. make changes to the non-recursive term RA or the start node. Or we try to further optimize the recursive part by changing the control flow of the strongly connected component of the CFG. The first method is called batching and will be examined in more detail in the following section, the second will be explored afterwards in Section 2.3.
Where exactly these two approaches take effect becomes clear if we look at a plan sketch of such translated UDFs. This sketch is shown in Figure 2.2. It shows the call of an arbitrary UDF within a query, the `UDF call environment`. Through translation, we get the recursive CTE `run`, which uses a `Recursive Union` node to compute the result from the initialization `RecInit` and the repeated evaluation of the recursive part `Rec` with a recursive self-reference to `run`. Batching now wants to optimize the left subtree `RecInit`, whereas the right subtree `Rec` can be optimized by using more than one self-reference.

```
WITH RECURSIVE run("rec?", res, current, goal, bbox, "track?") AS
  ((SELECT True,                                                          -- RA
           NULL :: box,
           start,
           NULL :: vec2,
           NULL :: box,
           False)
    UNION ALL
   (SELECT result.*
    FROM run, LATERAL
     ((SELECT ifresult7.*                                                 -- A0
       FROM LATERAL
         ((SELECT False,                                                  -- A0_0
                  bbox,
                  run.current,
                  run.goal,
                  run.bbox,
                  run."track?"
           WHERE  run."track?" AND current = goal)
          UNION ALL
          (SELECT ifresult12.*                                            -- A0_1
           FROM LATERAL (SELECT RTE0.*                                    -- Q1
                         FROM   squares AS RTE0
                         WHERE  RTE0.xy = current) AS square_4,
                LATERAL (SELECT RTE1.*                                    -- Q2
                         FROM   directions AS RTE1
                         WHERE  square_4.ll = RTE1.ll AND
                                square_4.lr = RTE1.lr AND
                                square_4.ul = RTE1.ul AND
                                square_4.ur = RTE1.ur) AS dir_4,
                LATERAL ((SELECT True,                                    -- A0_1_0
                                 NULL :: box,
                                 (current.x + dir_4.dir.x,
                                  current.y + dir_4.dir.y) :: vec2,
                                 current,
                                 bound_box(box(point(current.x,current.y)),
                                           box(point(current.x, current.y))),
                                 True
                          WHERE  (NOT run."track?" AND dir_4."track?"))
                           UNION ALL
                         ((SELECT True,                                   -- A0_1_1
                                 NULL :: box,
                                 (current.x + dir_4.dir.x,
                                  current.y + dir_4.dir.y) :: vec2,
                                 goal,
                                 bound_box(bbox, box(point(current.x, current.y))),
                                 run."track?"
                          WHERE  (NOT ((NOT run."track?" AND dir_4."track?"))) AND
                                 run."track?")
                           UNION ALL
                          (SELECT True,                                   -- A0_1_2
                                 NULL :: box,
                                 (current.x + dir_4.dir.x,
                                  current.y + dir_4.dir.y) :: vec2,
                                 goal,
                                 bbox,
                                 run."track?"
                          WHERE  (NOT ((NOT run."track?" AND dir_4."track?"))) AND NOT
                                 run."track?")
                         ))) AS ifresult12
           WHERE NOT (run."track?" AND current = goal))
         ) AS ifresult7)
     ) AS result
    WHERE run."rec?" = True))
SELECT run.res AS res
FROM   run
WHERE  run."rec?" = False
```

Listing 2.2: Compiled UDF bbox.

2.2 Idea of Batching

Now we assume that calls of our UDFs look something like in Listing 2.3. That is, we want to call the same UDF for multiple arguments that we extract from existing tables. Such a call behaviour is not untypical for DBMS. For example think of the administration of many products in a production company, which wants to check whether the margin for all products is still sufficient despite increased raw material prices by means of a UDF. For that the function must be used for all products of the assortment and will probably contain at least one query, which reads the current raw material prices from a separate table. Countless similar examples are conceivable that would lead to such a call context.

```
SELECT bbox((x, y) :: vec2)
FROM   generate_series(1, :invocations) AS i,
       LATERAL (VALUES (floor(     random() * 8  + i - i),
                       floor(1 + random() * (:iterations - 1) + i - i))) AS _(x,y);
```

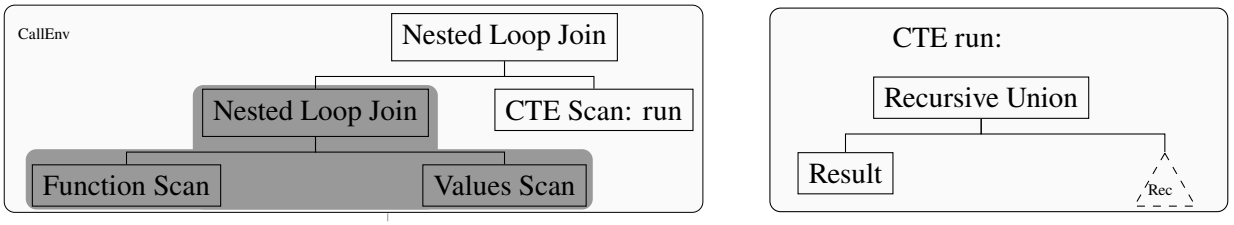
Listing 2.3: Call environment for UDF *bbox*.

In our artificial example are the parameters `invocations`, which controls how often the function is called and `iterations`, which represents the effort within a function call. In our example *bbox*, we set the `start` vector farther away from our object as we increase `iterations` and therefore require more executions of the loop to find the object and determine its bounding bbox.
The main idea of batching is to avoid these recurring function calls and to encode the `invocations` many start vectors (x, y) in one function call. If we take a look at our translated plain SQL query, this becomes even simpler and more obvious. Instead of many function calls encoding a single row in the working table `run`, we now want a single query execution that encodes all start vectors at once in `run`. One large table seems more natural to a DBMS than many tables with only one row. This idea leads to a small change for the non-recursive term (RA) to (RA_b) in Listing 2.2 as shown in Listing 2.4. There are no changes in the code for the recursive term.

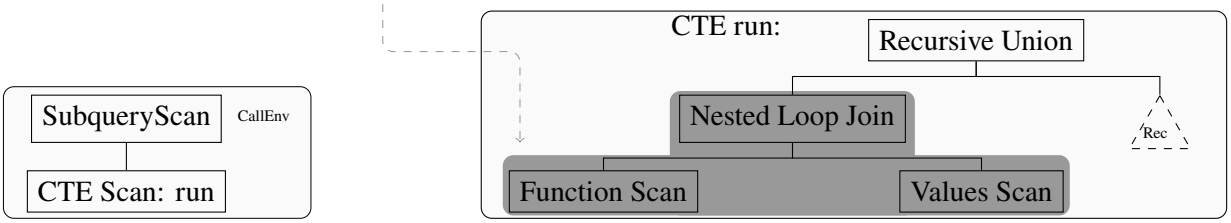
```
(SELECT True,                                                                    RA_b
        NULL :: box,
        start,
        NULL :: vec2,
        NULL :: box,
        False
 FROM   generate_series(1, :invocations) AS i,
        LATERAL (VALUES (floor(     random() * 8  + i - i),
                        floor(1 + random() * (:iterations - 1) + i - i))) AS _(x,y),
        LATERAL (SELECT ((x, y) :: vec2) AS start) AS __(start))
```

Listing 2.4: Batching of the compiled UDF *bbox*.

Such a natural way of batching does not exist for the PL/pgSQL version.
Now we look at the change of the plan by using batching for *bbox*. This is shown in Figure 2.3. We see the *Function Scan* of `generate_series` and the *Values Scan* to calculate `x` and `y`, which form the argument `start`. Without batching, the computation of the `start` vector takes place in the call environment and is fetched row by row from the `Nested Loop Join` in



(a) Call without batching similar to RA in Listing 2.2.



(b) Call with batching similar to RA_b in Listing 2.4.

Figure 2.3: Sketches of the plans for calling the compiled UDF *bbox* with and without batching.

the recursive CTE `run` by the *Result* node. Batching, however, pulls the complete computation inside the CTE `run` and initializes the recursion directly with all arguments. That means by batching the CTE `run` no longer has only one row per iteration, but several, as mentioned above.

For the calculations of the UDF *bbox*, this means that we no longer calculate the complete bounding box starting from one start vector and then consider the next start vector. Instead we calculate one step each for all start vectors according to the marching square algorithm. After moving all vectors by one pixel, we start a new iteration with the updated variable `current` instead of `start` and so on. If a calculation is completed (i.e. `rec? = false`), this is no longer looked at in the next step. Only the remaining ones are further calculated until finally all calculations are finished and the result of all computations is passed on.

As in this example, there is always a call environment that calculates the arguments of the UDF in tabular form and passes them via a `Nested Loop Join` to the CTE `run`. The table, which stores these arguments as rows, will immediately be coded into the `FROM` clause of the initial subquery `RecInit` of the recursive CTE to realize batching. In the plan sketch, only the greyed-out content for computing the arguments and any possible parent tree for computing parent queries ultimately adjusts. An example that structurally differs somewhat from *bbox* is *late*, which can also be found in the GitHub repository. This UDF computes a boolean and appears in the `WHERE` clause of the query. Nevertheless, batching works quite similarly here. We just have to move the UDF to the `FROM` clause and filter in the `WHERE` clause for `run.res` instead.

Now we can perform batching for all UDFs, but why should we do that? There are two major reasons: First, we hope that there are intelligent techniques in a DBMS that can make efficient use of large tables, giving an advantage over iterative one-row computations. On the other hand, we hope that plans can improve structurally and use smarter strategies like `Hash Joins` instead of `Nested Loop Joins`. We will later see that batching alone in PostgreSQL unfortunately does not bring any advantages, but if we additionally use multiple recursive references we can achieve helpful structural plan changes that bring a significant benefit. This is not surprising since PostgreSQLs `Nested Loop Joins` are executed iteratively and no form of parallelization takes place in the recursive part of a CTE. Therefore, in Chapter 3 we will look at another platform that uses *vectorization* and can benefit tremendously from

batching.
Up to this point, we have looked at the left-hand, non-recursive part of the `Recursive Union`, i.e., the initialization of the recursion `RecInit` resp. (RA), i.e., the query before the `UNION ALL` of a recursive CTE. In the following section, we want to investigate which possible improvements of the recursive right part of the `Recursive Union` we find, i.e. of `Rec` resp. the query after `UNION ALL`.

2.3 Idea of Multiple Recursive References

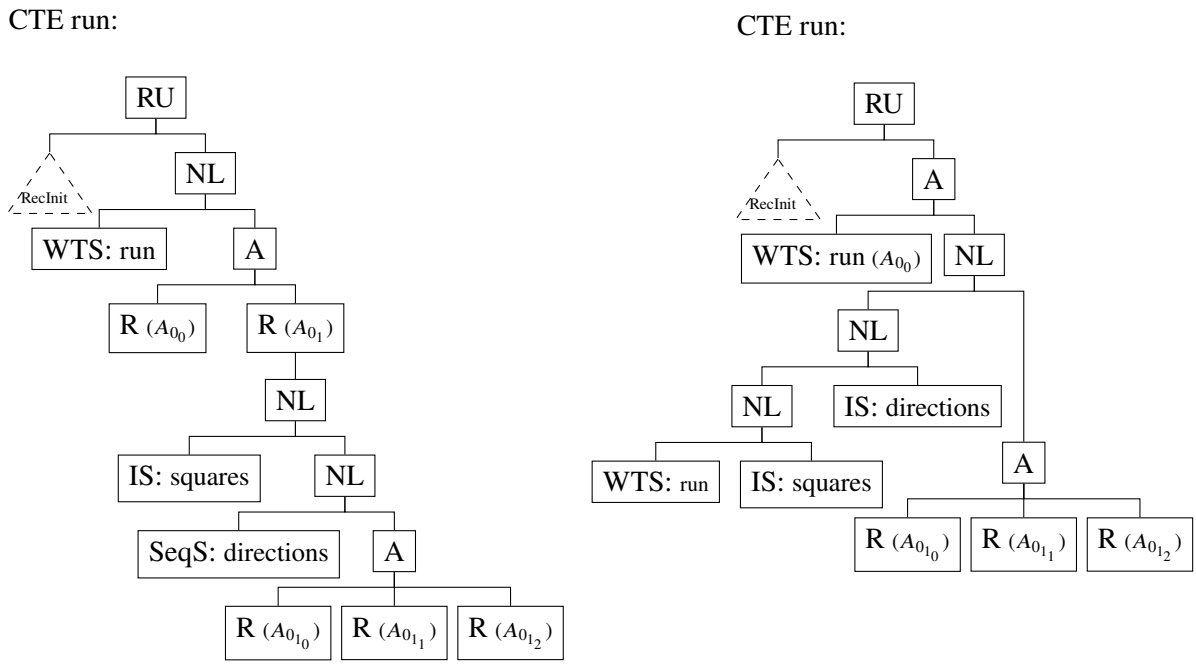
Nesting of `Nested Loop Joins` is never a smart idea in terms of efficiency. Therefore, in this section, we will present a technique to prevent this. First, we will look at how such a compiled UDF looks after the transformation described in [HG21] and, based on this, define rules that cause structural changes. So let us recapitulate the general template of a UDF, which can be seen in Listing 2.5. Here `Sb_i` stands for the `SELECT` clause of a block `b_i`, which corresponds to a function f_i that can be called by trampoline. Correspondingly, `Fb_i` is the `FROM` clause of such a block. A `WHERE` clause, on the other hand, does not exist because of the transformation rules. *Bbox*, for example, has only one such block labelled with `while1_head`, where the outer structure was manually optimized away.

```
WITH RECURSIVE run("rec?", label, res, ...) AS (
  SELECT true AS "rec?", 'fi' AS label, ...
    UNION ALL
  SELECT result.*
  FROM   run,
         LATERAL ((SELECT Sb_1
                   FROM   LATERAL Fb_1
                   WHERE  run.label = 'f1')
                     UNION ALL
                       ⋮
                     UNION ALL
                   (SELECT Sb_n
                    FROM   LATERAL Fb_n
                    WHERE  run.label = 'fn')) AS result
  WHERE run."rec?")
SELECT run.res FROM run WHERE NOT run."rec?";
```

Listing 2.5: SQL template of a compiled UDF. Cf. [HG21, Figure 13]

This query structure is too complex for PostgreSQL's optimizer to produce efficient plans that use join operators other than the `Nested Loop Join`. Thus, the plan of the recursive part of the translated version of our example *bbox* (see Figure 2.4a) looks accordingly: A deep nesting of `Nested Loop Joins`, all of which yield the worst expected performance. This example is still quite simple, while larger examples with more complex program flow can easily nest numerous `Nested Loop Joins`.
One quickly notices the keyword `LATERAL`[2], which is essential for the general template of Listing 2.5. However, `LATERAL` leads to an iterative row by row behaviour for evaluating the actual recursive work. Also, this keyword takes a lot of freedom away from the optimizer, since the evaluation order is restricted. For now it may not matter if we have a behaviour

[2] cf. https://www.postgresql.org/docs/14/queries-table-expressions.html



(a) Plan sketch for the CTE run from original compiled UDF *bbox*.

(b) Plan sketch for the CTE run from UDF *bbox* using two recursive references.

Figure 2.4: Plan modification by using two recursive references to the working table `run`. Abbreviations: Recursive Union (RU), Append (A), WorkTable Scan (WTS), Nested Loop Join (NL), Index Scan (IS), Sequential Scan (SeqS), Result (R).

similar to a nested for-loop if we have only one row in `run` anyway. But at least with regard to the previously discussed batching, it becomes relevant that we do not restrict the evaluation in such a way. At this point, the idea comes into play to use more than one reference to our working table `run` so that each block gets its own reference and thus the `LATERAL` join can be avoided. However, this is not yet provided for in any PostgreSQL version and leads to an error. Therefore, we have to apply a simple trick by starting the recursive part of the query with `(WITH runCTE AS (SELECT * FROM run))`. Thus, only one reference to the working table `run` is needed to define the CTE `runCTE`, to which we may subsequently reference as many times as we need to. To avoid this workaround, we use a patch by Denis Hirn that allows multiple references to the working table. However, these two techniques do not show any significant differences in their performance.
This idea allows rewriting the template so that we get the new version of Listing 2.6. This is valid code only, if we use the patch. Otherwise it must be adjusted using the trick mentioned above. However, many examples – including *bbox* – do not have multiple labels at all after block inlining, and yet we observe that the DBMS is unable to create a more efficient plan. So we need to look further into the individual blocks and prevent nesting of `ifresult` tables, which also stand out because of the keyword `LATERAL`. This nesting is caused by branches in the program flow within a loop construct. The general grammar for creating such a block using multiple label-level references can be found in Listing 2.7. Here `Q` is a query and `p` is an arbitrary predicate. The first production rule cannot be optimized further, as `run` is accessed directly in the `SELECT` clause and therefore we cannot move this reference to subqueries. The second production rule, on the other hand, is the union of two blocks that can have their own reference to `run` instead of a common one. Looking at a block in its context as in Listing 2.8, we get the transformation as seen in Listing 2.9 using one more recursive reference. Here `u` is

```
WITH RECURSIVE run("rec?", label, res, ...) AS (
  SELECT true AS "rec?", 'label_i' AS label, ...
    UNION ALL
  (SELECT Sb_1
   FROM   run, LATERAL Fb_1
   WHERE  run."rec?" AND run.label = 'label_1'
     UNION ALL
       ⋮
     UNION ALL
   SELECT Sb_n
   FROM   run, LATERAL Fb_n
   WHERE  run."rec?" AND run.label = 'label_n')
SELECT run.res FROM run WHERE NOT run."rec?";
```

Listing 2.6: SQL template of a compiled UDF using multiple recursive references.

```
b → (SELECT "rec?", "label", "res", "col_1", ..., "col_n"
     FROM run, [LATERAL Q AS let('var_1', ..., 'var_n'),])
       | (SELECT ifresult_p.*
          FROM  run, [LATERAL Q AS let('var_1', ..., 'var_n'),]
                LATERAL (b WHERE p
                           UNION ALL
                         b WHERE NOT p) AS ifresult_p)
```

Listing 2.7: Grammar describing a block b.

a predicate and `Fif_p` is a general join tree. It is immediately noticeable that we duplicate code thereby. A copy of `Fif_p` is needed for both blocks since this is mostly code for the computation of `p`. So we need systematics or heuristics that tell us if it makes sense to take this step. We will take care of this in the next section. We now want to apply this idea to our example *bbox* and get in a first step Listing 2.10, where the code parts that have not changed to Listing 2.2 are only sketched. It should be noted while we have optimized away the only label in the code, we still need to perform the first step in an adjusted form to achieve the shape of the template in Listing 2.6. How this change affects the plan is shown in Figure 2.4b. We eliminate the topmost `Nested Loop Join` by this optimization and, so to speak, throttle the `WorkTable Scan` into both branches of the first `Append` node. This way A_{0_0} is calculated by its own `WorkTable Scan` and we can use an `Index Scan` instead of a `Sequential Scan` for `directions`. If additionally batching is used, the tables `directions` and `squares` are hashed and the associated `Nested Loop Joins` become `Hash Joins`, which allows us to increase performance significantly. As you can see it is possible to use this idea a second time and eliminate `ifresult12`. In this case, however, we would have to perform the relatively expensive calculations of `square_4` and `dir_4` three times. However, we would only get one large append node with four children and no more nesting.

Before we now turn to the systematics mentioned earlier, we want to illuminate this method from another angle. Instead of doing transformations in the query, we can examine it directly in the CFG. Quite similar to the trampolined style used in [HG21] to control the individual labels, we can now control through a central node, here called `branching`, which path is to be taken within the CFG and thus successively unfold the branches in the CFG. This node thus acts analogously to the `trampoline` node. In fact, we can even combine the `trampoline`

```
(SELECT ifresult_p.*
 FROM   run, LATERAL Fif_p,
        LATERAL (SELECT Sb1 FROM LATERAL Fb1 WHERE p
                   UNION ALL
                 SELECT Sb2 FROM LATERAL Fb2 WHERE NOT p) AS ifresult_p
 WHERE  run."rec?" AND u)
```

Listing 2.8: SQL template of a block following the second production rule in the context of the general UDF template after using multiple recursive references.

```
(SELECT Sb1
 FROM   run, LATERAL Fif_p, LATERAL Fb1
 WHERE  run."rec?" AND u AND p)
   UNION ALL
(SELECT Sb2
 FROM   run, LATERAL Fif_p, LATERAL Fb2
 WHERE  run."rec?" AND u AND NOT p)
```

Listing 2.9: Transformation of the block template 2.8 using one more recursive reference.

node and the `branching` node into one node. To do this, we assume binary branches using only simple if-statements. If we check in the beginning instead of somewhere in the program which branch of the if-statement is taken, we *extract* this path from the existing CFG and delete the associated if-statement that made it possible. In doing so, all nodes involved are duplicated. Finally, all predicates that determine this particular path are conjunctively combined and passed into the central node with the goal of gradually reducing the cyclomatic complexity (also known as *McCabe metric* [McC76]) of the subgraphs to which the `branching` node now refers. Thus, we can gradually further partition each subgraph with more than one possible path. As seen before, code can be duplicated in this way and gradually the predicates defining the possible subgraph are expanded conjunctively. How the first level A_0 from Listing 2.2 is unfolded into its two child-levels A_{0_0} and A_{0_1} by extracting the first branch is illustrated by the CFG in Figure 2.5. It corresponds to Listing 2.10.
In fact, this way one can imagine controlling the program flow even more generally than before. For example, the above fixed template transformations does not allow us to extract the path ($A_{0_{1_2}}$) first. This is simply because we would have to copy the branch in `while1_body` in each case, which we want to avoid. So our procedure is the same as extracting single paths starting with the first possible branch, which also always ensures that the variables used by the predicate are assigned correctly.

2.3.1 Defining a Heuristic

Up to this point, we have seen how we can recursively reduce the nesting of `Nested Loop Joins`. However, we have also seen that in doing so, calculations may have to be performed multiple times. The question arises, when to take such a step and use more recursive references and at which point the additional effort by repeated computations is greater than the expected performance gain by more simplified plans. The main idea we follow here is that we want to have as few `Nested Loop Joins` as possible, and thus as flat plans as possible without having to perform the same computations multiple times.

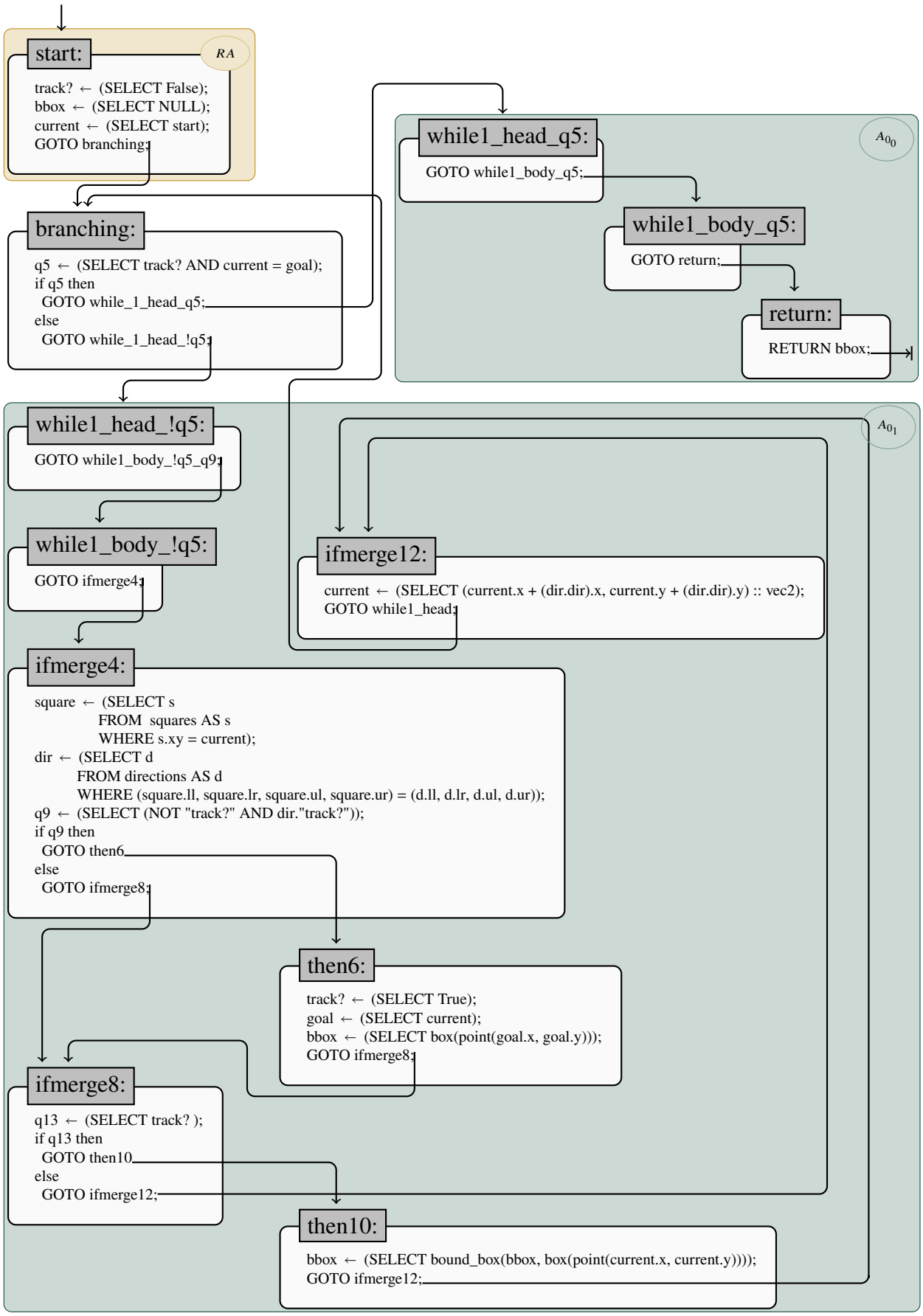
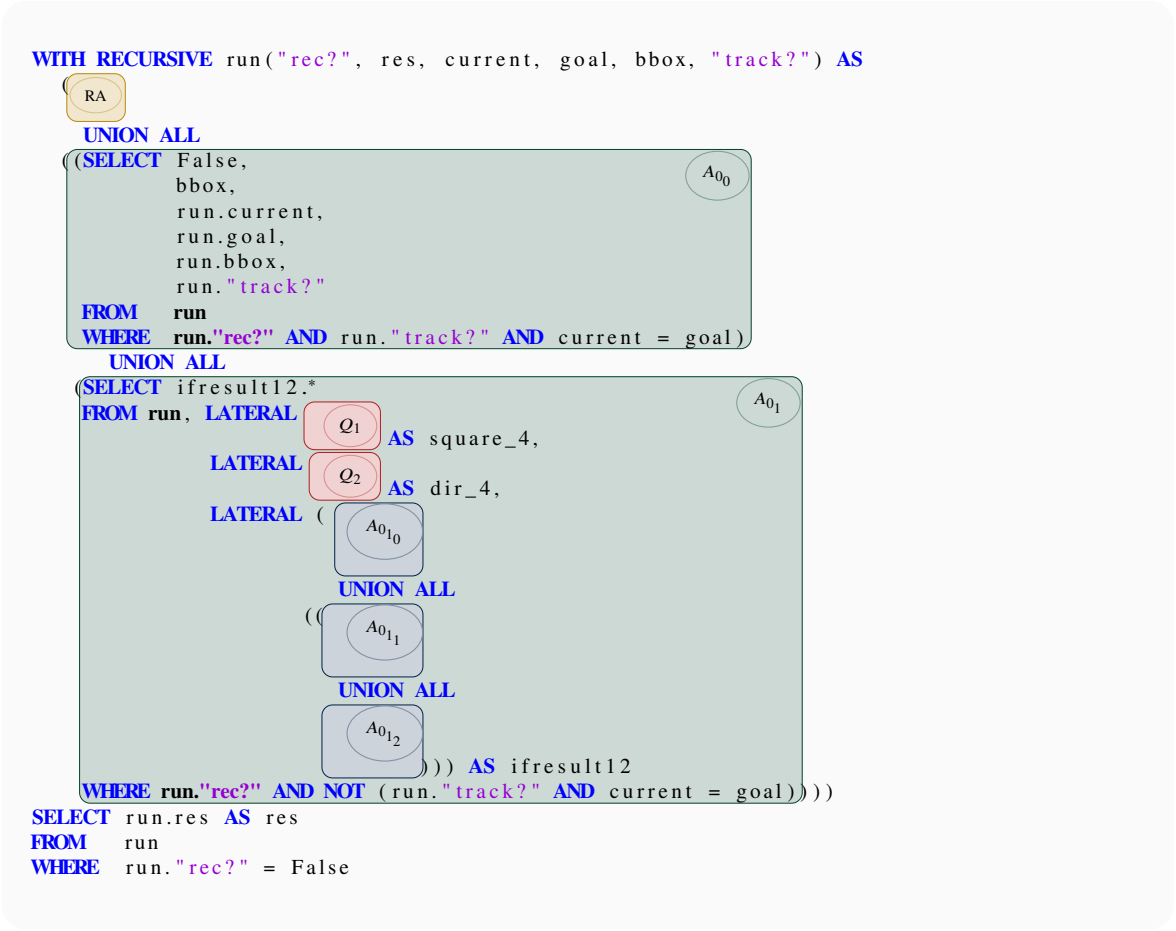


Figure 2.5: CFG for UDF *bbox* after *path extraction*.



Listing 2.10: Compiled UDF *bbox* using two recursive references.

It is clear that we can only insert more references at the block level if we have previously performed the transformation at the label level. Since each label block in the template is independent of the others, no copies of computations are made. There should be no disadvantages if we perform independent computations based on the whole table instead of individual rows. Therefore we always apply this transformation which leads to the template in Listing 2.6. It gives us the starting position on which we can perform the individual block transformations.

The transformation within a block from Listing 2.8 to Listing 2.9, on the other hand, we do not want to simply always perform as well. While this would result in plans that are as flat as possible, and in the best case even have just one big `Append` node with `WorkTable Scans` under it, we can just duplicate the computations that go into `Fif_p` to compute our predicate. For *bbox* it was already mentioned that we triple the subtree with two `Nested Loop Joins`, two `Index Scans` and one `WorkTable Scan` when performing the transformation a second time. However, if these `Index Scans` are now very expensive, our optimization potential is significantly less than the additional cost incurred. This illustrates that depending on `Fif_p`, we have to decide whether to continue recursively with the transformation or leave it at this level. To do so, we want to have a look at what are possible initial situations. On the one hand there are very simple, partly constant variable assignments like `(SELECT 42 AS "magic_number") AS let("magic_number")`. These are partly even inlined and should therefore not keep us from our transformation. Even minor calculations like equality checks of two simple expressions do not need a significant runtime and can be executed twice as

often without any problems. It only becomes problematic when we have to compute queries that access whole tables, like `square_4` and `dir_4` in our example. Also, when functions are called they may have a very long runtime and are better not executed twice. An example of such a function is unnesting of arrays as in the UDF *schedule*. Compiling queries based on external tables results in using the *RTE* characters as seen in our example *bbox*, in which case function calls lead to the characters *RTFunc* in the query. So we can easily detect them and avoid copies of them by aborting the recursive transformation.
However, we need to consider one exception. If we have only one label and the associated block starts with a join tree containing possible expensive queries, each sub-block within this block will probably always have to use the defined and computed variables. Nevertheless, we do not want to have multiple copies of this query, of course. Here it is possible – under consideration of the above-mentioned trick of introducing the `runCTE` – to carry out the calculation centrally and to use nevertheless several recursive references. To do so, we calculate the needed variables already in the definition of `runCTE` so we can access columns of our CTE instead of referencing the previous join tree. In general, this gives us the transformation of the query into a form like in Listing 2.11.

```
WITH RECURSIVE run("rec?", label, res, ...)
  ((SELECT true, ...
    FROM   ...)
    UNION ALL
  (WITH runCTE AS (SELECT run.*, Fif_p.* FROM run, LATERAL Fif_p)
   (SELECT Sb1
    FROM   runCTE AS run, LATERAL Fb1
    WHERE  run."rec?" AND u AND p)
      UNION ALL
   (SELECT Sb2
    FROM   runCTE AS run, LATERAL Fb2
    WHERE  run."rec?" AND u AND NOT p)))
SELECT run.res FROM run WHERE NOT "rec?";
```

Listing 2.11: Extension of the working table with variables of the join tree `Fif_p`.

Note that it makes a real difference to calculate these columns only for `runCTE` instead of adding them to `run`. This way the variables are calculated centrally for all rows before the next iteration step, instead of having to calculate them again in a row-by-row style in the individual blocks for the following step.
In principle, this extension of the working table could be introduced at arbitrary places and of course also with more than one label. However, it is difficult to predict whether this would cause unnecessary overhead by calculating variables for rows that never use them on their way through the corresponding loop. So it would be a matter of studying how often which path would be taken in the CFG. By doing so, we could use cost parameters to estimate which variables should be included in `runCTE` and which ones would unnecessarily add a lot of overhead. Therefore, we leave our heuristics simple to that effect and only choose this extension if and only if we have one label and its corresponding block starts immediately with complex variables.
For *bbox*, this small set of rules is very appropriate since we would go no further than we showed in Listing 2.10 and thus avoid the copies of `Index Scan`s. To which extent the heuristic turns out to be optimal in our further examples can be found in Chapter 4.1.6.
At this point we want to clarify the last open question, namely which compiled versions we want to fall back on when we apply our transformations. The compiler we use has the option

oanf, which inlines all blocks as much as possible and thus provides fewer labels overall. In fact, block inlining creates structures that can lead to copies of code when introducing multiple references later on. Nevertheless, it has been seen that in our examples the maximally inlined versions perform better than all transformations we get without the *oanf* option, so this flag is always set in our investigations. Thus we do not differ in the initial situation from the versions as they can be found in the already mentioned GitHub repository.

2.4 Migration to a Newer Version of PostgreSQL

The paper [HG21] on which this work is based wrote its UDFs for PostgreSQL 11.3. Since this version is now somewhat outdated, we adopted their work to PostgreSQL 14.1. The results we discuss in Chapter 4.1 were then generated under the new version.
The first thing you notice during the migration is that the PL/pgSQL UDFs tend to become faster, so improvements for imperative programming style are already implemented. Another new feature is that *JIT*[3] is used by default, which we deactivated in our measurements to avoid overhead with partly short query runtimes. Likewise a `memoize`[4] node is added, which mostly does not seem to have much impact, but can lead to problems in comparing different approaches, as we will see later.
Relevant for us is that we cannot use `LEFT OUTER JOIN`s to prevent unwanted compiler optimizations, as it was possible in the 11 version. It turns out that by replacing all `(LATERAL LEFT OUTER JOIN ... ON TRUE)`s with a simple `LATERAL` we generally fare better and improve the readability of the compiled code enormously. While manual inlining of simple variables in the older version had an influence on planning and running time, this happens mostly by itself in the current version. Also predicates, for example, are now directly simplified by logical transformations. For us, this means that we have to worry less about manually optimizing the automatically compiled UDFs in the new version. However, this automatic inlining goes definitely too far insofar as effortful calculations are copied dozens of times, e.g. the UDF *margin* has 40 copies of a `Index Scan` with following aggregation while only one would be necessary. These copies are found in many other examples and are a consequence of the aggressive inlining. As before, by effortful calculations we mean those which have dependencies on other tables or functions of the DBMS, i.e. which are marked in the code by the keywords `RTE` respectively `RTfunc`.
These copies must be prevented to achieve the best runtime. To avoid making changes to the internals of PostgreSQL that would then again only run on our specific version, we again use a trick to outsmart the optimizer. It turns out that if we introduce a CTE, the optimizer can no longer inline it. Specifically, this means that if we have a code snippet like the one in Listing 2.12, we rewrite it into the form of Listing 2.13. We do the same with `RTfunc`. Of course, we only want to do this if there is a concrete risk of multiple copies, otherwise, we would unnecessarily limit the optimizer. Actually, this problem is exactly given if we access variables computed in this way at least twice, either directly or indirectly. *Indirectly* refers to using a variable more than once, which depends in some way on a variable that is used in such a `RTE` respectively `RTfunc` query. This systematic approach leads to optimal results for all the examples we considered. The appropriate time to use this trick is after introducing multiple references. This way, even the special case where only one block has to be split further despite

[3] https://www.postgresql.org/docs/14/jit.html
[4] https://www.postgresql.org/docs/14/runtime-config-query.html#GUC-ENABLE-MEMOIZE

```
... LATERAL (SELECT ... RTE.* ...
             FROM   ... some_table AS RTE ...
             WHERE  p) AS "var" ...
```

Listing 2.12: Code snippet of a effortful calculation.

```
... LATERAL (WITH "var_CTE" AS (SELECT ... RTE.* ...
                                FROM   ... some_table AS RTE ...
                                WHERE  p)
             (SELECT * FROM "var_CTE")) AS "var" ...
```

Listing 2.13: Code snippet of a effortful calculation after injection of a CTE.

complex calculations can be handled correctly. All other transformations are independent of each other.
Another conceivable approach, which can also prevent such unwanted inlining, is the already mentioned extension of the CTE `run` by the corresponding variables, which would be calculated several times. However, since this leads to optimal runtimes only in special cases we apply the other variant.

We end our methods chapter on PostgreSQL by summarising the developed heuristics for the current version 14.1 in compact form:

(i) Compile the PL/pgSQL version with *oanf* flag. Possibly perform further manual optimizations like eliminating unnecessary columns in `run`.

(ii) Replace `LEFT OUTER JOIN`s.

(iii) Place multiple recursive references:

  (a) If the uppermost `ifresult` is through an elaborate predicate, introduce a CTE augmented by this predicate.

  (b) Recursively assign a own recursive reference for all subqueries, if they are only distinguished by a simple predicate.

(iv) Use CTEs to prevent multiple code copies of expensively computed predicates.

Exploring DuckDB 3

In this chapter we will test our ideas with another DBMS. We are especially interested in *vectorization* since we hope to achieve great improvements for batching with this technique. Unfortunately, batching alone does not have the desired effect for PostgreSQL, since we still have to work with `Nested Loop Joins`. We therefore choose DuckDB[1] for comparison.

3.1 Introduction and Methodology

DuckDB contains a *columnar-vectorized query execution engine*, which allows us to achieve exactly what we hope to do with batching. Instead of processing values one by one, multiple values are processed in the shape of a vector. We will be able to benefit enormously from this, as we will see in more detail in Chapter 4.2.

3.1.1 Preparing UDFs and Avoidance of Limitations

None of the versions created for PostgreSQL can directly be run using DuckDB. Since this is a very early DBMS, many features that we use from PostgreSQL are still missing. First of all, we cannot declare our own types, so for example any query that wants to use a vector of type `vec2` cannot be used. In this case, we have to avoid these type-declarations and work with e.g. separate x and y values instead. Operators defined for types must also always be replaced in the code by their definition. This leads to a lot of small local changes, which we cannot avoid in the current version anyway. But there are even more profound missing features, including no support for `LATERAL Joins` and no `UNIONs` in the recursive part of a CTE.
It is possible to work around `LATERAL Joins` by using multiple CTEs. The unsupported code in Listing 3.1, for example, is rewritten to valid DuckDB code in Listing 3.2. In doing so, we must keep in mind that by forming the Cartesian product, we lose the membership associated with a single row as in a `LATERAL Join`, requiring an additional column `id` for `run`, which must be uniquely initialized in the recursion initialization. With respect to these, columns and calculations can be uniquely identified across all tables. Thus the `id` acts as a kind of *foreign key* and together with CTEs allows us to replace the `LATERAL Join`.

```
SELECT ...
FROM   run, LATERAL table1 AS t, LATERAL table2 AS s
WHERE  ...;
```

Listing 3.1: General structure of our `FROM` clauses using `LATERAL Joins`.

[1] https://duckdb.org/

```
WITH run AS (SELECT * FROM run), t AS (SELECT run.id, table1.* FROM run, table1), s AS
    (SELECT run.id, table2.* FROM run, t, table2 WHERE run.id = t.id)
SELECT ...
FROM   run, t, s
WHERE run.id = t.id AND run.id = s.id ...;
```

Listing 3.2: Avoiding `LATERAL Joins` using CTEs and `ids`.

We still need to find a way to avoid `UNION ALL` in the recursive part. To do so, we use case distinctions by means of `CASE WHEN`. The starting point for this is the complete recursive transformation of a UDF into its individual paths each with its own reference. That means we do not use the heuristic we defined before, but always perform all possible transformation steps. As a result we have the union of individual subqueries whose `SELECT` clause contains only the variable assignments of `run` and whose `WHERE` clause contains the predicate that precisely determines the path in the CFG. The `FROM` clause contains CTEs that simulate arbitrary calculations via `LATERAL`. The general template looks something like Listing 3.3.

```
WITH run AS (...), CTE1 AS (...), ...
  (SELECT run.id, true, label_i, res_i,...
   FROM   run, CTE1, ...
   WHERE  run.id = CTE1.id AND ... AND p1)
     UNION ALL
       ⋮
     UNION ALL
  (SELECT run.id, false, label_j, res_j, ...
   FROM   run, CTEm, ...
   WHERE  run.id = CTEm.id AND ... AND pn)
```

Listing 3.3: Template for the recursive part of the CTE `run` after complete recursive transformation.

Subsequently, we can make case distinctions in each individual column with respect to all possible paths. We thus get the shape of Listing 3.4. Naturally, we can also very easily combine some blocks here by using the `ELSE` statement. For instance, the `rec?` column in particular usually has the assignment `true`, so we can explicitly write all other cases down and combine most cases in one (`ELSE true`).
We can replace the imperative PL/pgSQL style very easily with a canonical Python script. Analogous to PostgreSQL, we get a procedural language style version, as well as versions with and without batching. However, due to the limitations and the structural change made thereby, we do not have an analogon for the use of multiple recursive references, instead we always have only one translated version.

3.2 Future Work and Possible Applications for PostgreSQL

The transformations necessary due to missing features can also be done for PostgreSQL. Especially due to the possible collapse of many single blocks into one `ELSE` statement we could benefit from this representation. A few examples were already examined and compared with our previous work. It showed very similar runtimes and concise, compact plans that do

```
WITH run AS (...), CTE1 AS (...), ...
  (SELECT run.id,
          CASE WHEN p1 THEN true
               WHEN p2 ...
                  ⋮
               WHEN pn THEN false
          END ,
          CASE WHEN p1 THEN label_i
                ⋮
               WHEN pn THEN label_j
          END ,
          ⋮
  FROM  run, CTE1, ....
  WHERE run.id = CTE1.id AND run.id = CTE2.id AND ...)
```

Listing 3.4: DuckDB supported template avoiding `LATERAL` and `UNION ALL`.

not use `Nested Loop Join`s but are entirely based on `Hash` and `Merge Join`s. Therefore, a more detailed investigation in this direction seems appropriate.
It is also worth implementing missing features for DuckDB to simplify testing of already existing UDFs. Additionally, the influence of different APIs on the runtimes should be investigated more closely. Instead of the Python API used here, the native C++ variant might be better.

Experiments and Results 4

In this chapter, we will discuss how well the techniques presented earlier perform on our examples and how to generate comparable measurements. For this purpose, we will first discuss the results of PostgreSQL in detail, before presenting the results for DuckDB.

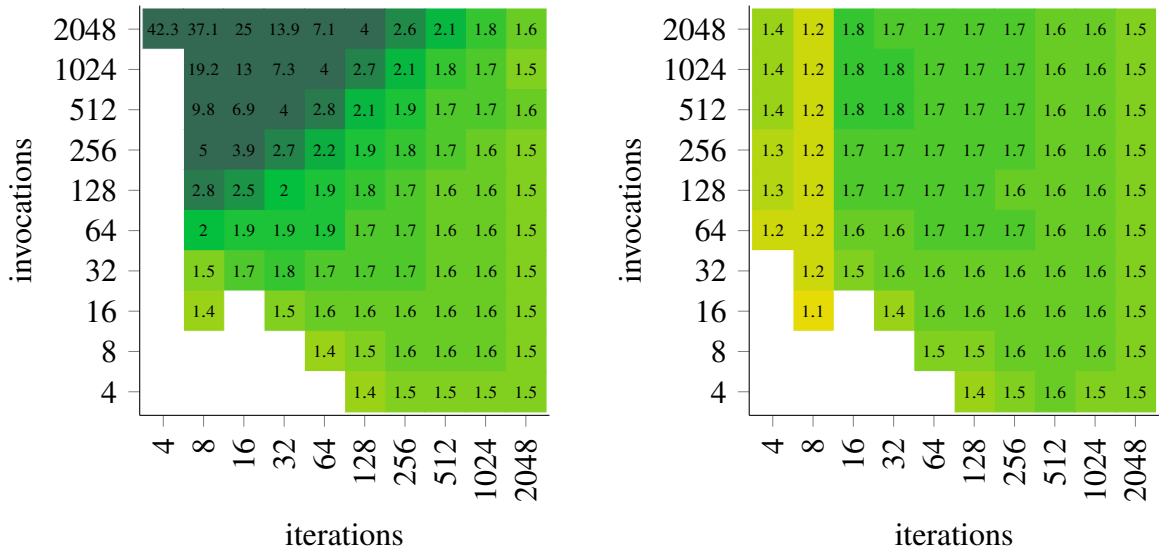
4.1 PostgreSQL

Let's start with PostgreSQL as in the fundamental work [HG21]. Here we have the possibility to test the performance of the mentioned batching and the multiple references, as well as the combination of them. Furthermore, we have the possibility to use the construct `WITH ITERATE` by a patch to reduce memory and copy issues.

4.1.1 Setup

We will start with examining the general test setup. The final measurements were performed on the department's server. It runs on Ubuntu 22.04 with 512GB RAM (at the time of testing). Two AMD EPYC 7402 processors are available.
As mentioned before, we use the PostgreSQL 14.1 version. This version includes the patch by Denis Hirn so we use several references to the working table by default. The JIT option was not compiled here and therefore turned off. The `work_mem` parameter is set to 10GB unless explicitly mentioned otherwise. No other changes were made to the default configuration. In particular, we use `WITH RECURSIVE` and not `WITH ITERATE`, unless otherwise stated.
Measurements were performed consecutively and repeated five times. The median of these measurements was then calculated to eliminate outliers. In this way, the runtimes were determined independently for all variants and then compared. Only values greater than 20ms were taken into account, since even shorter runtimes contain too much variance and are not representative. The measurements were parametrized by `invocations` (resp. `sf` for TPC-H problems), i.e. the number of rows in the working table `run`, and by `iterations` , i.e. the complexity inside the function call. TPC-H[1] is a database benchmark with an industry-wide relevance. The UDFs *items, late, margin, packing, savings* and *sched* adopted from [HG21] are based on it.
For TPC-H problems, we tested the scaling factors (sf) 0.01, 0.05, 0.1, 0.5, and 1.0, each of which was further decomposed in complexity by limiting it to $\frac{\texttt{iterations}}{5}$ ($\texttt{iterations} \in \{1, \ldots, 5\}$) of the columns. For all other problems, we chose the limit of `iterations` and `invocations` by a suitable multiple of 2, such that the maximum running time of a UDF was approximately between three minutes and one hour depending of the UDFs complexity.
We can visualize these values using heatmaps like in [HG21, Figure 18], over which we can

[1] https://www.tpc.org/tpch/



(a) With duplicate removal.

(b) Without duplicate removal.

Figure 4.1: Heatmaps for UDF *bbox* illustrating the speedup after transformation of *pl* to *tr*.

subsequently calculate the mean and standard deviation to obtain a compact representation of the average performance. Such a heatmap can be found for our running example *bbox* in Figure 4.1.

4.1.2 Introducing our UDFs

We used the examples of [HG21] to test our techniques. We were interested in the runtime of the underlying PL/pgSQL version – hereafter called *pl* – as well as the translations without using additional techniques (*tr*), the batched translations (*tr-batching*), those with multiple references (*tr-multRef*) and those translations that combine both (*tr-multRef-batching*). Since we did a version upgrade of PostgreSQL, all runtimes and speedups might have changed, so we also had to make sure to re-specify the performance gain from translation alone. To do this, compare [HG21, Table 1] with Figure 4.2. The table for the old version uses the center of the heatmap in each case, which means that no exact comparisons can be made. Slower for all parameters – i.e. to the left of the dashed line or speedup less than 1 – are again only *ray* and *sheet*. They performed even a bit worse compared to the previous version, which is mainly due to the fact that PL/pgSQL became significantly faster in some cases. In the new version, *march* and *bbox* were clearly faster, and *vm* was somewhat worse. All other values lie within the range of the standard deviation. Let's take a closer look at the partly very good values for *bbox*. The heatmap in Figure 4.1a shows that extreme values occurred mainly at the boundary points with few iterations. Most of the heatmap contains values between 1.5 and 2, which is quite similar to the already known value of the previous work. Looking at the plans more closely, we can easily explain the extremely good values in the upper left corner. There were no plan changes, but PostgreSQL was smart enough to recognizes multiple occurrences of function arguments. The corresponding results were then not recalculated each time, but only once per argument. So it happens that instead of 2048 rows with 16 `iterations` we only calculated 120 unique parameters. This feature is exactly provided by the `memoize` node, which is new in PostgreSQL 14. Due to the structure of *bbox*, such duplicates are more likely to occur when the `iterations` are small and, of course, when the `invocations`

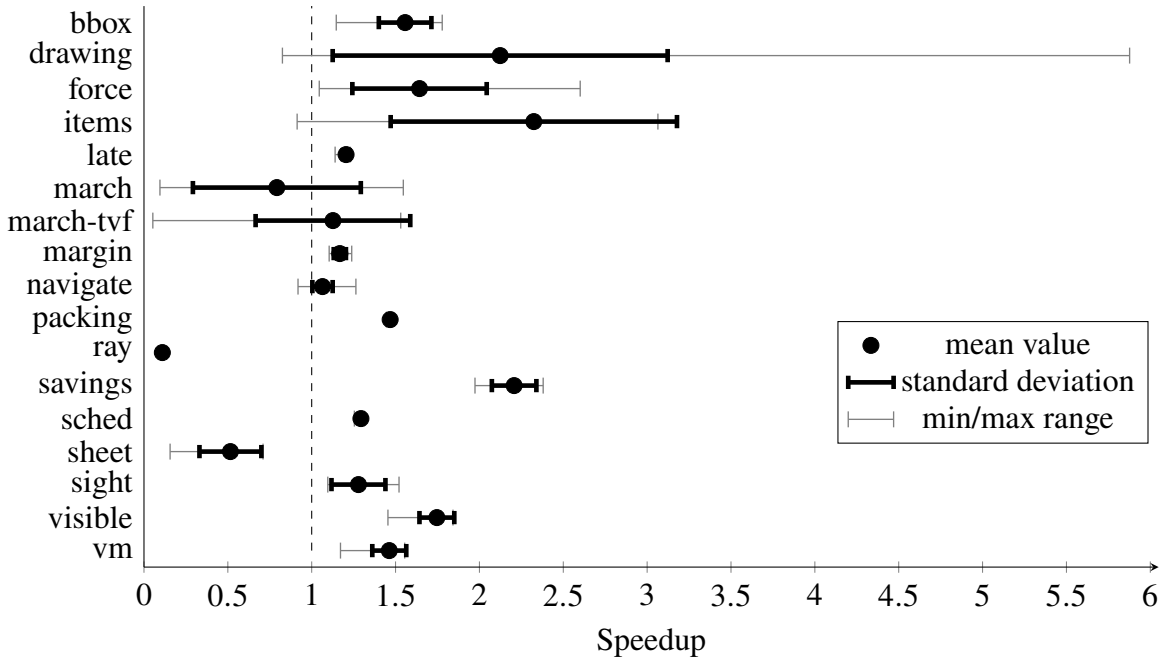


Figure 4.2: Speedup after transformation of *pl* to *tr* for all UDFs based on mean value and standard deviation over all instances. The range from minimum to maximum speedup of each UDF is included.

are large. This explains the breakouts perfectly. The question remains whether we want to consider this optimization as a feature or a bug. In principle, it makes a lot of sense, but it is absolutely impractical for our analysis since neither *pl* nor the batched variants use such duplicate avoidance. Therefore, we made sure to include a unique id in the working table `run`. For the three UDFs (*bbox, march* and *march-tvf*) which caused problems, this was simply done by using the `i` from `generate_series` which was computed anyway. One could also go along and disable `memoize`, but this would also prevent optimization potential elsewhere. Figure 4.1b shows the heatmap that is obtained without such duplicate avoidance strategies. It can be seen that all outliers have been eliminated and we have very constant results across all points. This led to comparable results across versions. If we would not do this, the next step would show a minimum speedup of 0.03 when comparing *tr* with *tr-batching*, i.e. an enormous slowdown, which makes batching look worse than it really is.
In the following, we always refer to the versions without duplicate avoidance. That way it is ensured that we do not lose the benefit of the translation compared to PL/pgSQL due to the version change and the changes that come along with it.

Defining a New Example

In the process of finding an optimal heuristic, case distinctions on enumerative types, such as those found in *vm* with respect to `opcode`, turn out to be particularly problematic. In this case, we have to do more recursive steps than one would expect from the other examples. However, using the special case of the CTE extension, we can see near-optimal results. To verify this, another example was defined that also has these case distinctions on enumerative types to show. *Drawing* is inspired by *KTurtle*[2] and is intended to aggregate simple drawing commands and

[2] https://edu.kde.org/kturtle/

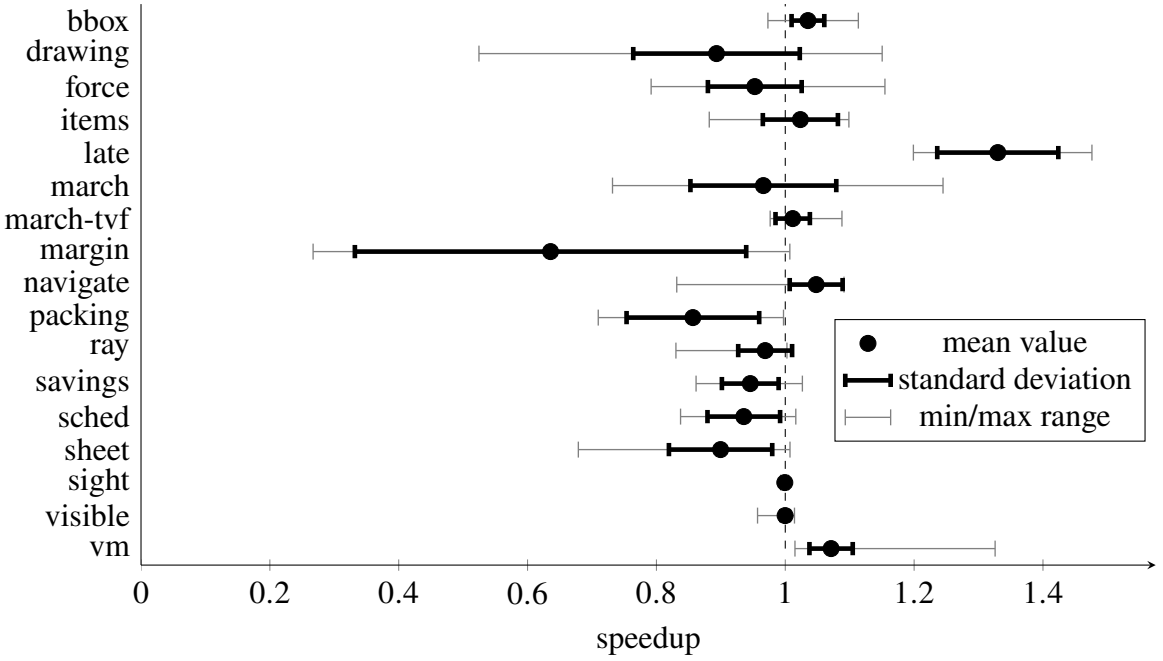
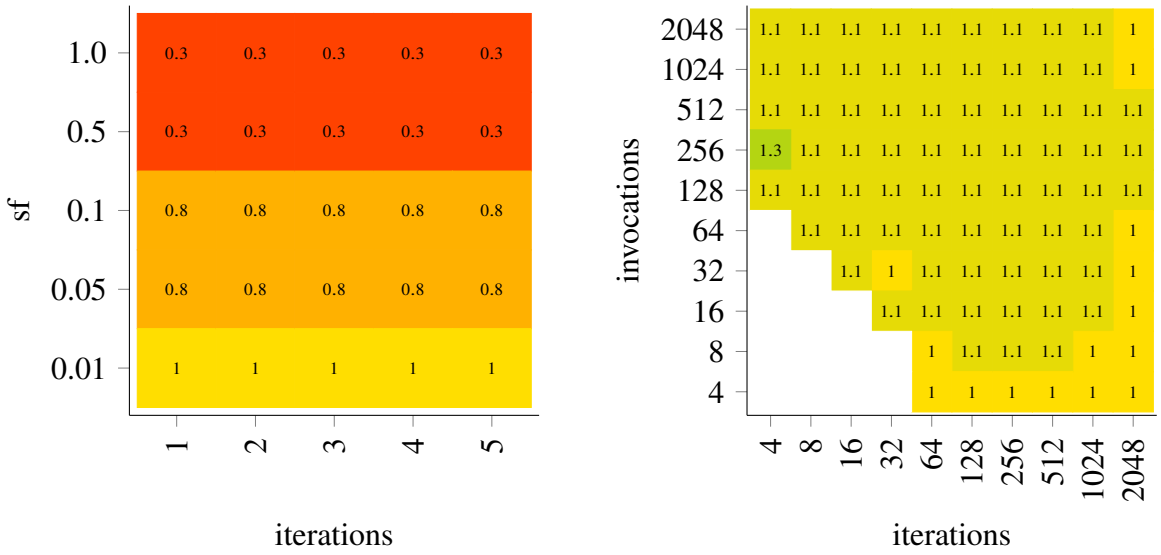


Figure 4.3: Speedup after transformation of *tr* to *tr-batching* for all UDFs.

combine them into a whole object. In this work, the drawing direction was divided into eight possible directions by an enumerative type `dir`. This example helps us argue that our heuristic can also handle the identified problem case well. If we do not aggregate the directions in our example but add them, we get the variation *navigate*, which only computes the endpoint given a starting point and finitely many instructions.

4.1.3 Using Batching

Let's turn to the first technique – batching as introduced in Section 2.2. It is important that we allocated sufficient memory. As `run` can become a potentially very large table, too little `work_mem` will require an extremely large amount of read and write work in the form of temporary buffers. It becomes particulary problematic, if we provide as little memory that hash tables cannot fit in it. We used enough working memory (10GB) to prevent runtime abnormalities due to temporary written buffers. As indicated earlier, we could not show the desired performance boosts from batching due to the `Nested Loop Join`s that PostgreSQL uses. As a point of comparison, we took the translations *tr* and then looked at the speedup or slowdown by introducing batching in Figure 4.3.
Most results range around one, so there was actually very little change in the runtime. Problematically, *margin* could degrade significantly. The whole heatmap is shown in Figure 4.4a. Clearly, we can see that the results worse as the TPC-H size increases. This was mainly because batching uses an additional `Sequential Scan` over `part`. *Late* performed particularly well. Together with *items*, these were the only examples where the function call occurred within the `WHERE` clause. This allowed *late* to perform further optimizations not directly related to batching. For example, a `Hash Join` was used to further process the result of `run` instead of a `Nested Loop Join`. This explains this outlier.
Another example that seemed to perform relatively well was *vm*. However, in Figure 4.4b we see that the good value was found singularly among many very similar values around 1.1.



(a) UDF *margin*.

(b) UDF *vm*.

Figure 4.4: Some heatmaps illustrating the speedup after transformation of *tr* to *tr-batching*.

In conclusion, batching alone should not be used under PostgreSQL, as the risk of getting significantly worse runtimes is much greater than the minimal chance of small improvements. When deciding to take `meomize` as a feature and choosing a clever data representation without unnecessary columns, batching can even perform extremely worse as already mentioned. But our efforts are not completely lost, as can be seen in combination with multiple references and especially with DuckDB.

4.1.4 Using Multiple Recursive References

Now that we have studied the non-recursive term, we will look at the performance of the transformation of the recursive term. To do so, we use the heuristic we defined in Section 2.3.1. In Figure 4.5 the possible speedup of all examined UDFs is shown as usual. The picture here is quite similar to the one above for batching. Most examples hardly changed and have the tendency to get slightly worse. The problem is that we cannot get out of the iterative row-by-row behaviour, which contributes significantly to the runtime. Let's now take a closer look at some noticeable examples.
First of all, the extremely bad minimal values of *bbox, march* and *march-tvf* are interesting. Also noticeable is that their results match almost perfectly. The corresponding heatmaps can be found in Figure 4.6. These algorithms are all based on the *marching square algorithm* and so it is not entirely unexpected that they behave very much the same. First, we see that the vast majority of the parameter configurations did not change much and we only observe extremely bad values for certain `iterations`. We will now explain these in more detail. Recalling back, we already know that we could take a recursive step on *bbox* before the effortful predicate `dir_4` appeared. So is likewise for the other two algorithms. When looking at the plans, it is obvious that a plan change regarding the calculation of this predicate is crucial for the red band of poor performance. Here PostgreSQL tries to be clever with 16 and 32 `iterations` and uses a `Hash Join`, but as usual, the smaller table is used as hash, which here is `run`. Of course, this makes no sense since `run` is a single row and therefore we only get a poorly performing `Nested Loop Join`. Further, the hash table cannot be

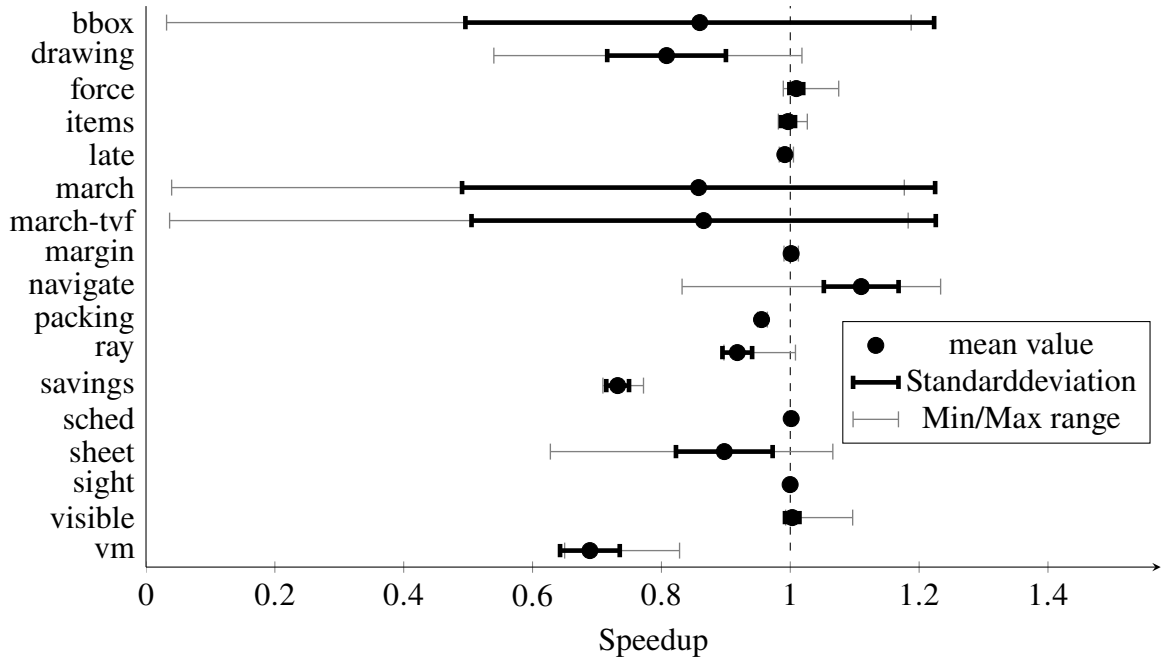


Figure 4.5: Speedup after transformation of *tr* to *tr-multRef* for all UDFs.

reused since `run` changes in every iteration. As a small side effect of this, we had to use a `Sequential Scan` on `squares` instead of a `Index Scan`. Summa summarum, we got an inefficient computation of `dir_4` in these cases. All other parameter configurations showed the expected and already explained behaviour. Thus, the topmost `Nested Loop Join` was eliminated and the `WorkTable Scan` was propagated into both branches, which means no putative optimizations occurred for `dir_4`.The problem came from a very specific size of the `map` table, which occurs for exactly 16 or 32 `iterations`.

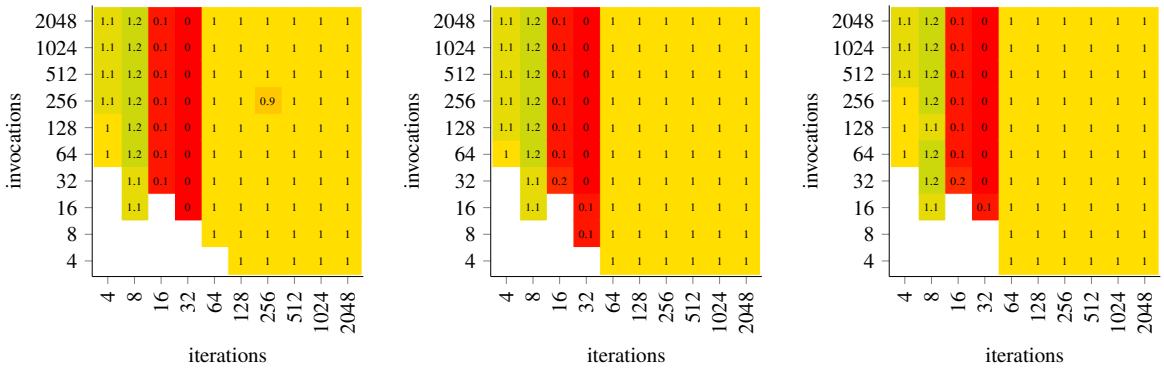
Vm also performed noticeably poorly. Except for very few outliers, we observed a speedup of about $^2/_3$. The problem is quite analogous to introducing `Hash Join`s which want to hash `run`. So again an inappropriate join operator was used, which additionally prevented a useful operator, such as the `Bitmap Index Scan` in this case.

Savings performed also consistently bad with a speedup of about $^3/_4$. Nothing unpredictable took place here. The plans also do not indicate why the results necessarily worse here. Only some `memoize` nodes arose from our transformation, which may take more time in their initialization than they can save later.

Sheet also caused a speedup around 0.85, except for a few outliers. Instead of a many-nested plan, we got a very flat plan with many single `Worktable Scan`s. Here we cannot say where the slight degradation came from. However, a tendency that we can improve with increasing effort is visible.

Otherwise, unsurprisingly, the results remained almost unchanged. For us, this means that multiple references alone can already generate `Hash Joins`, but these mistakenly always use `run` as a hash. This may cause our UDFs to even get worse. For most, there is no relevant change, since the iterative behaviour of `Nested Loop Join`s hardly changed due to our transformation.

Following, we will combine both techniques with the goal to improve the `Hash Join`s, which have already been created, by hashing the more suitable table.



(a) UDF *bbox*. (b) UDF *march*. (c) UDF *march-tvf*.

Figure 4.6: Some heatmaps illustrating the speedup after transformation of *tr* to *tr-multRef*.

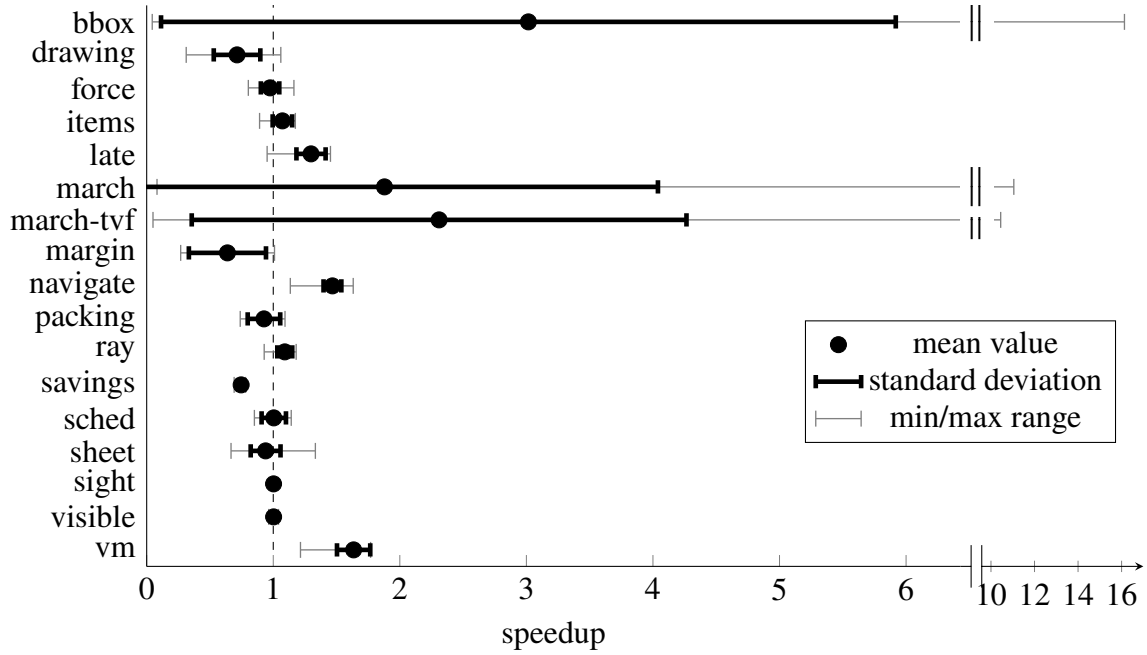


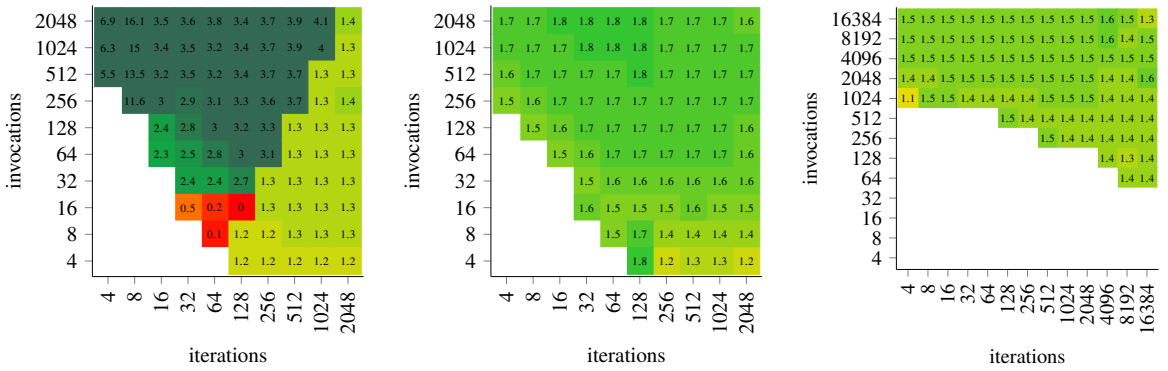
Figure 4.7: Speedup after transformation of *tr* to *tr-multRef-batching* for all UDFs.

4.1.5 Combining Batching with Multiple Recursive References

As usual, we also compactly summarize the results using a combination of both techniques in Figure 4.7. First, we noticed that only *savings* performed worse in all points due to our transformations. For every other UDF, there was at least one parameter configuration, so that we were at least on par with *tr*. The mean value of *drawing, margin, packing* and *sheet* show that those UDFs also decrease in performance. This results in 5 out of 17 examples that cannot benefit from this techniques. These UDFs were actually already worse with *tr-batching* and *tr-multRef* than *tr*.

In contrast, there were speedups of up to 16.1 with *bbox*. In a nutshell, the transformation was significant and worthwhile for *bbox, march-tvf, march, vm, navigate* and *late*.

For now, let's take a closer look at *bbox* and its heatmap in Figure 4.8a. As with *tr-multRef*, we had partial `Hash Join`s present. This time, however, not with `run` as hash, but `squares` is hashed once and then reused. So we actually benefit a lot from this `Hash Join` compared to a simple `Nested Loop Join`. The jumps in the heatmap from less performance gain with low



(a) UDF *bbox*. (b) UDF *vm*. (c) UDF *navigate*.

Figure 4.8: Some heatmaps illustrating the speedup after transformation of *tr* to *tr-multRef-batching*.

`invocations` relative to `iterations` to more performance gain can be explained by just such a planchange from `Nested Loop Joins` to `Hash Joins`. For example, the parameter configuration (512, 256) uses a `Hash Join`, whereas (512, 128) uses a `Nested Loop Join`. As mentioned earlier, *march* and *march-tvf* are similar algorithms and the performance gain in all three examples can be explained by the way the common predicate `dir_4` is computed. Therefore, the plans for all three examples look very similar here as well, i.e., with the same jumps in the heat maps due to the use of `Hash Joins`.
In contrast, the heatmap for *vm* in Figure 4.8b presents a much more homogeneous picture. The relatively constant factor of about 1.7 also comes from the use of `Hash Joins`. As with *bbox*, we also had the problem of hashing the wrong table for *tr-multRef*, which is solved here. This lead to a better performance of the `Hash Join`.
When no longer `run`, but the table `program` is used as hash is only due to the `invocations`, because the smaller table is used for hashing. In our case, `program` had 18 rows, which caused all the factors with more `invocations` to profit from the more performant `Hash Join`. For *bbox*, the size of `squares` depends on the `iterations`, so such a behaviour arises there as well. Thus, for *vm* there is a visible jump with respect to exceeding a constant number of `invocations` due to the constant sized table `program`. For *bbox*, the size of `squares` is approximately $\texttt{iterations}^2$, making the area of the performance jump similar to a parabola. The good performance of *late*, on the other hand, can only be attributed to batching, which has already been successful. Multiple references were neither an advantage nor a disadvantageous. The performance gain with *navigate* cannot be traced back to special plan changes. Here, both batching and the use of multiple references were helpful, although only the expected behaviour occurred here. It may be noticeable that we managed to turn a deep tree of `result` nodes into a very shallow plan with only `CTE Scans` over run.
In conclusion, however, it is not only the comparison to the translation *tr* that is interesting, but in particular how well we competed against the original PL/pgSQL version. These values are presented in Figure 4.9. It can be seen that the performance increased in comparison to the starting point with the exceptions of *margin, ray* and *sheet*. Factors between 2 and 5 are quite realistic. Thus, we can safely apply the transformations using our heuristics as a standard technique.

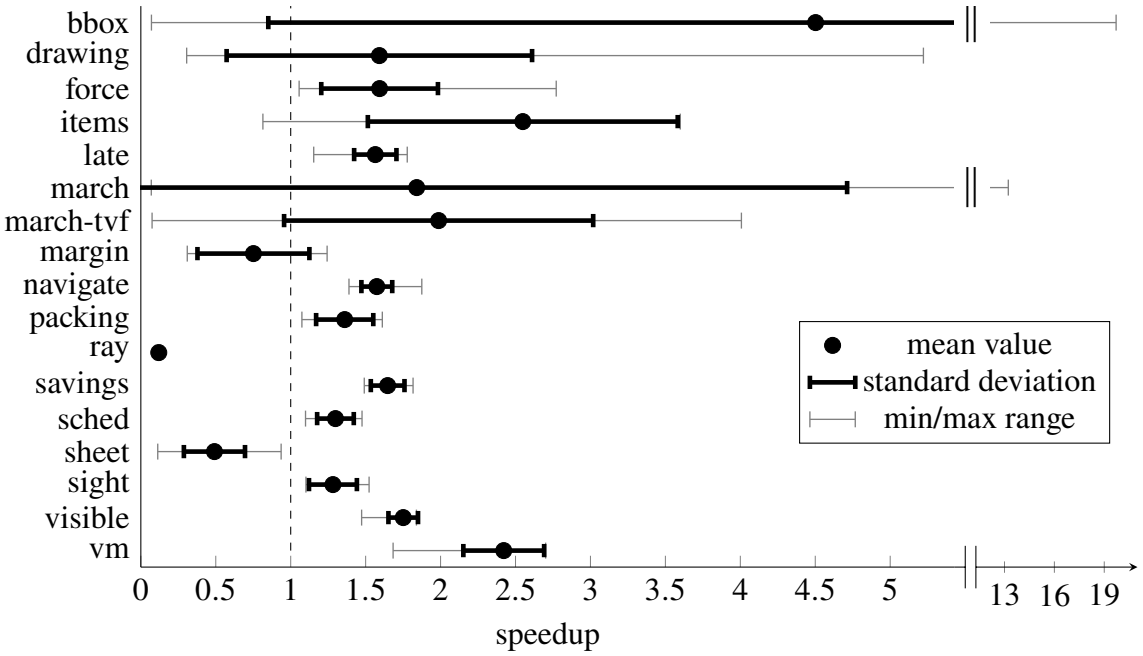
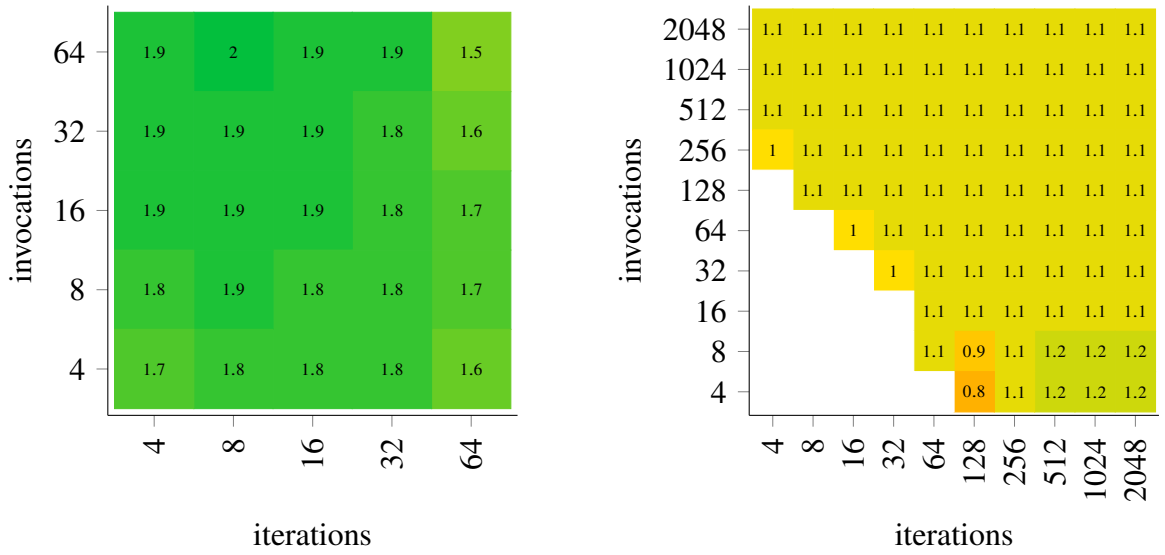


Figure 4.9: Speedup after transformation of *pl* to *tr-multRef-batching* for all UDFs.

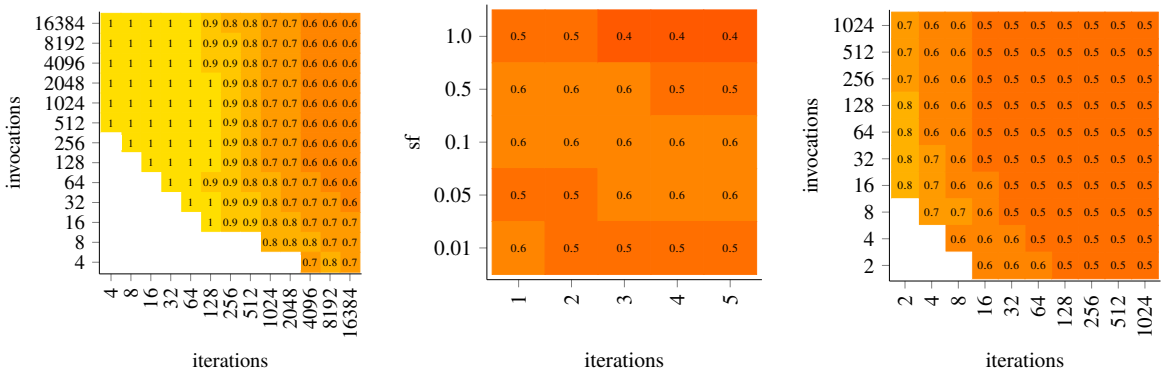
4.1.6 Optimality of the Heuristic

The question remains whether our results represent the maximum that is possible or only a fraction of the possible performance boost. In other words, we investigate how well our heuristic from Section 2.3.1 really works and in which cases is significantly more achievable. To answer this question, all possible versions were tested for each example with several usual parameters. That means both the setting of CTEs to avoid code coupling in PostgreSQL 14 and every possible recursive descent when setting multiple references were tested. Also, the extensions of `runCTE` by chosen predicates were examined frequently. It was also considered that different initial points regarding the labels are possible by the compiler option *oanf*, whereby all our UDFs turned out to be better with *oanf* than without, even after further manual optimizations. In the end, we also tested a few examples regarding the style we get from the necessary transformations for DuckDB in Chapter 3.
Let's first look at the necessary steps to achieve the migration from PostgreSQL 11 to version 14. In particular, here the `LATERAL LEFT OUTER JOIN` was replaced by a simple `LATERAL JOIN`. There were numerous degradations due to various partial plan copies. After setting the CTEs with respect to the described heuristics, we can thus get all versions except *ray* at least equally fast. However, both *ray* and *vm* were slightly faster without CTEs. These and subsequent observations were incorporated into a hand-optimized version, which can be seen in Figure 4.10. All other examples were (nearly) optimal for version 14 after transformation. Let's see if our heuristic performed the optimal number of recursive transformations regarding the multiple references. Here there are at least better versions in the examples *late, march* and *sheet*. However, these were in a very small, negligible range. *Ray*, on the other hand, performed significantly better when using a recursive reference for each label instead of following our heuristics. These deviating optimal levels were incorporated into the manual versions in Figure 4.10. It can be seen that for the highly complex UDF *ray* there may be almost another doubling in speedup. However, the possible further improvement with *vm* turned out to be quite small.



(a) UDF *ray*. (b) UDF *vm*.

Figure 4.10: Heatmaps illustrating the speedup from *tr-multRef-batching* with our heuristic compared to *tr-multRef-batching* with manual optimization.

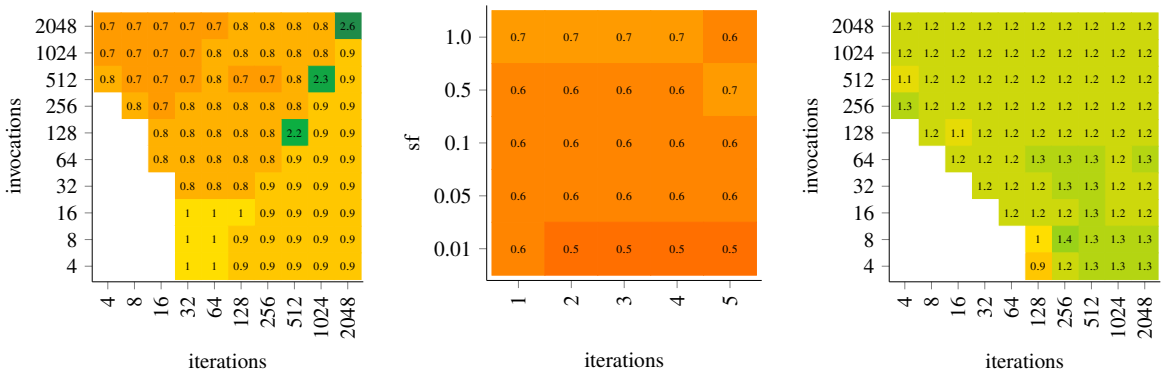


(a) UDF *force*. (b) UDF *packing*. (c) UDF *visible*.

Figure 4.11: Heatmaps illustrating the speedup from *tr-multRef-batching* with our heuristic compared to *tr-multRef-batching* with all possible recursive transformations.

To emphasize the importance of suitable heuristics, we will now take a look at what can happen if all the putative optimizations are stubbornly applied right to the end. For this purpose, Figure 4.11 shows three selected UDFs for which the recursive transformation was applied to the end. You can see here that you can easily get runtimes twice as long and thus destroy any advantage of a translation.

Finally, let's look at the style of DuckDB via case distinctions in the `SELECT` clause. The analysed examples are collected in Figure 4.12. The plans that emerge from these versions are structurally very elegant and avoid `Nested Loop Join`s as much as possible. Instead, they rely on `Merge Join`s and `Hash Join`s. As we can see from these examples, the performance of *vm* can be extended further than with our transformations and heuristics. Accordingly, a closer examination might be beneficial.



(a) UDF *bbox*. (b) UDF *packing*. (c) UDF *vm*.

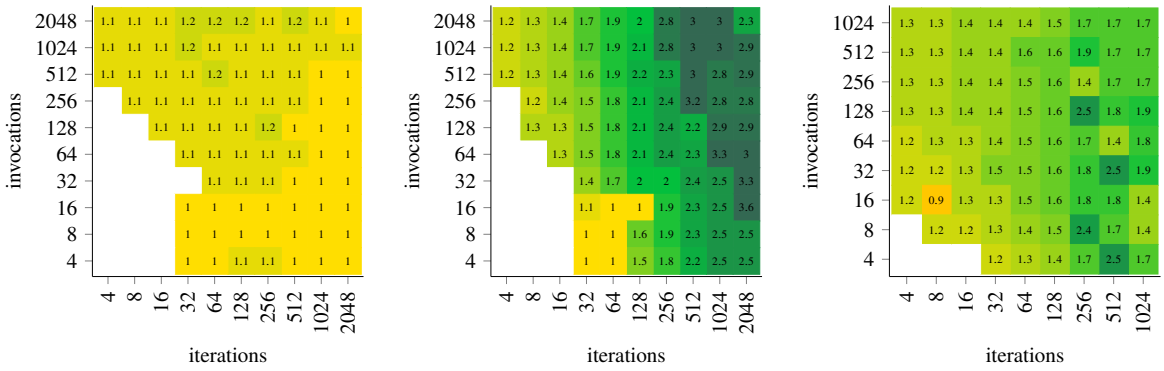
Figure 4.12: Heatmaps illustrating the speedup from *tr-multRef-batching* with our heuristic compared to *tr-multRef-batching* in the case-distinction-style of DuckDB.

4.1.7 Further Potential Using `WITH ITERATE`

Up to this point, we have tried all possible external optimizations that PostgreSQL allows. The next step is to adapt the internals of the system to our needs. This is exactly what is done by Denis Hirn's patch implementing `WITH ITERATE`. To do this, we first need to understand how `WITH RECURSIVE` behaves in our case. All intermediate results are written to a temporary table and after recursion, almost all rows are sorted out by the filter `run."rec?" = True`. These rows are at most interesting for debugging or something similar and can be eliminated completely most times. A row should only be passed to the `UNION` operator for further saving if the column `rec` contains the value `true`. This way a lot of I/O operations can be saved. Depending on the number and type of the columns, there is a table width and depending on the required number of `iterations`, a table length. This approach works especially if at least one of the two quantities is relatively large. This explains, for example, why *march* benefits significantly from this, but *march-tvf* does not. This is because only *march* needs a column of type vec2-array, which increases the table width enormously.
The details of `WITH ITERATE` are described more precisely in [HG21, Chap. 4.1], as well as in [DHG19, HG20, PTH+17].
Since this feature is not available as standard, we have not built on it yet, so that anyone can reproduce our results without further patches. Nevertheless, we will have a look at the further potential using selected UDFs. The important thing to note is that while we can partially compensate for missing `wok_mem` with `WITH ITERATE`, this feature can sometimes show significant improvements even with enough memory. Mostly, the possible improvements are in the single-digit percentage area, but there are also significantly better versions, as can be seen in Figure 4.13. The important thing is that none of the performance gains is due to a lack of memory. As mentioned earlier, *march* has a large table width and a relatively large table length, which can account for the speedup. In *sheet*, we also have a lot of columns containing an array, which also means a lot of writes. Since *bbox* has comparatively small columns and few iterations, the speedup here is rather low as well.

4.1.8 Limitations

Finally, let's take a look at the limitations of our approach. Those who have carefully followed our choice of the UDFs may have already seen that we have not included *markov*. This is



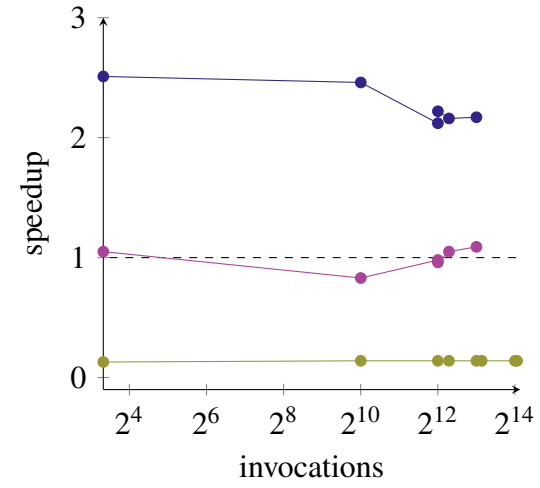
(a) UDF *bbox*. (b) UDF *march*. (c) UDF *sheet*.

Figure 4.13: Heatmaps illustrating the speedup from *tr-multRef-batching* using `WITH RECURSIVE` compared to *tr-multRef-batching* using `WITH ITERATE`.

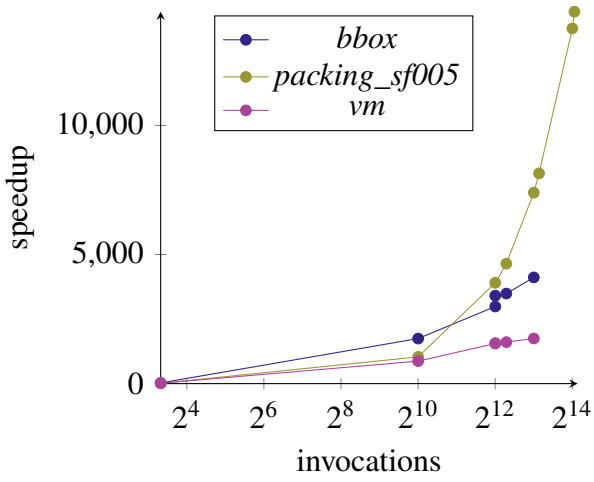
because we do not get comparable, even partly wrong behaviour by our transformation here. This is explained by our first limitation by itself: Randomized algorithms.
If we use calls like `random()`, we face several problems at once. On the one hand, we change the calling order and thus the sequence of our pseudorandom numbers by batching. This leads to changed results, which cannot be prevented by setting a seed, whereby no direct comparability is given anymore. However, one can of course argue with the fact that with randomized algorithms it does not matter to us in principle if we change the sequence of pseudorandom numbers. Here there is by multiple testing with different seeds and averaging over the results the possibility to make the versions comparable again. On the other hand, there is still the problem of unwanted optimization by PostgreSQL, so that depending on the context a `random()` is evaluated once and afterwards returned as a constant value. Thus, we eliminate any randomness in the algorithm and get completely unwanted behaviour, up to infinite loops. Therefore, it is recommended to test only non-randomized algorithms with our methods under the current PostgreSQL version.
Additionally, the limitations of the underlying paper [HG21] still apply. Among those, we can only look at `STABLE` UDFs, i.e. those that the database state cannot modify, since we translate into a read-only SQL query.

4.2 DuckDB

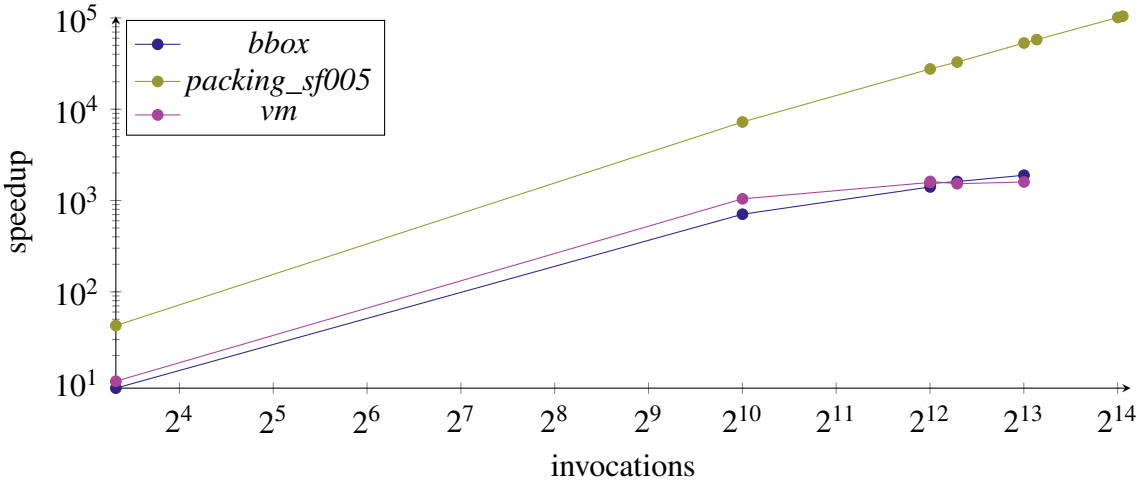
After the unsatisfying results with PostgreSQL, we will take a look at our second DBMS. DuckDB produced much more promising results and show that batching is very useful after all. All measurements were done on the same hardware as for PostgreSQL. The Python API with python version 3.10.4 and DuckDB 0.3.3 was used. *Bbox, packing* and *vm* were examined, as they are relatively easy to rewrite since they use only a few user defined types and no unknown operators. We kept the `iterations` or the TPC-H instance as a constant in each case and examined only the dependency on the `invocations`. The *pl* version was implemented as a python program with database queries. The *tr* and *tr-batching* versions, on the other hand, were a prepared statement. Figures 4.14a and 4.14b show the results with *pl* as reference. The x-axes are scaled logarithmically. It is immediately noticeable how poor our *tr* translation of *packing* was. This is due to the fact that by falling back on CTEs to avoid `LATERAL` among other things, we have obtained a naturally batched version here, which we subsequently have



(a) Transformation of *pl* to *tr*.



(b) Transformation of *pl* to *tr-batching*.



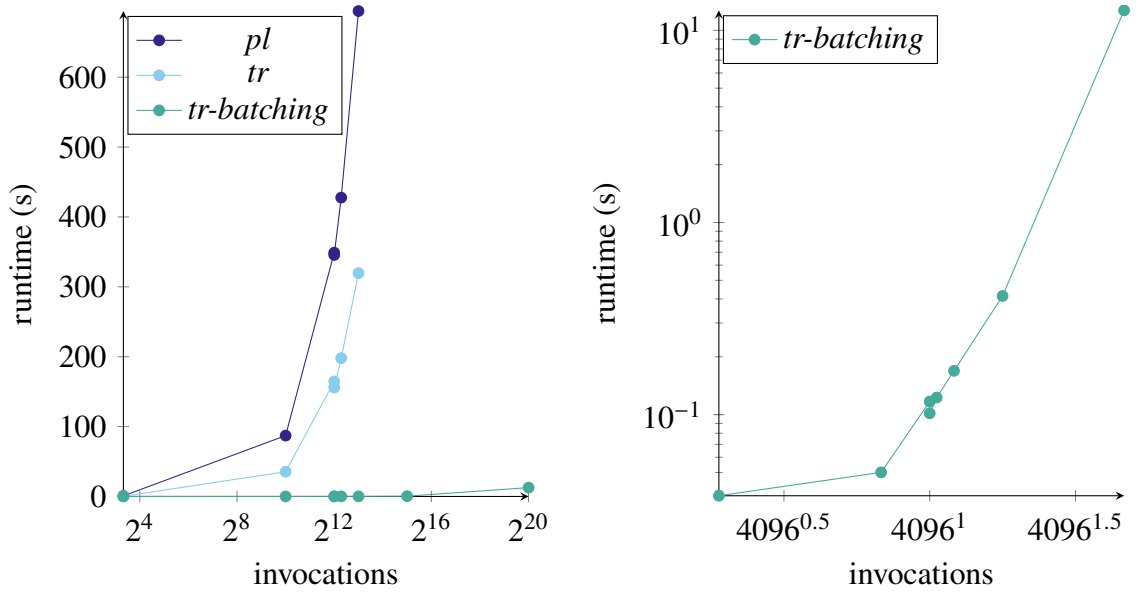
(c) Transformation of *tr* to *tr-batching*. Both axes logarithmic scaled.

Figure 4.14: Speedup of selected UDFs.

to artificially pick apart into individual rows again. From this, you can already see how natural batching actually is. Interestingly, *bbox* had a similar speedup as in PostgreSQL, *vm* did barely change.

Although *packing* was significantly faster than its translation in the *pl* version, we could observe a huge speedup using *tr-batching*. The slow *tr* version of *packing* also explains why a speedup of up to over 100 000 can be achieved in Figure 4.14c. However, we also obtained factors of over 1000 for the other UDFs, which clearly highlights the advantage of batching. Even though we did not sufficiently investigate to what extent our UDFs are optimally written, either way, a very clear advantage can always be expected.

A logarithmic close to linear growth of the respect to the `invocations` can be seen in Figure 4.14b and Figure 4.14c. To illustrate this in more detail, let's look at the evolution of the runtime with respect to *bbox*. This is shown in Figure 4.15a with logarithmic x-axis. It can be seen very nicely here that both *pl*, and *tr* behave roughly linearly, as we would expect. I.e. if you have twice as many calls, you need about twice as long. In contrast, *tr-batching* looks almost logarithmic. However, if you change the angle and look at the larger scale (see Figure 4.15b), you can see that above 4096 an almost linear growth starts. This reflects the vector size that DuckDB uses in its calculations.



(a) With x axes logarithmic scaled.

(b) With boths axes logarithmic scaled.

Figure 4.15: The runtime of *bbox* for different versions depending on `invocations`.

Overall, this results in excellent performance gains that are significant even for very small batch sizes and can be increased almost arbitrarily. For example, even 10 `invocations` results in a speedup for all UDFs by a factor of above 8 when comparing *tr* with *tr-batching*. Looking at the hardware during the computation, this advantage is also clearly visible from the CPU usage. It can be seen that one or two cores are fully utilized and many further show a lower strongly changing utilization. Thus it is clear to see that here parallel tasks are worked on, in contrast to PostgreSQL, where only one core is used at a time.
We have already discussed the limitations due to missing features in Chapter 3.

Conclusion & Future Work 5

We now conclude this work, briefly summarizing the results and looking at where there is further potential for future research.
We have seen quite clearly that batching alone does not result in any real improvement for PostgreSQL unless optimizations happen to appear in the plans that do not arise directly from batching. Similarly, the use of multiple recursive references alone is not recommended, as we can only get little improvement from this, but also significant losses caused by `Hash Joinss` with incorrectly chosen tables to be hashed. One can now try in future work to change the internal criteria for a `Hash Join` in a recursive CTE, so that never a single row working table will be hashed. Here we have to develop a heuristic, to determine when it makes sense to use the working table as hash and when it makes more sense to use tables that do not change during the iteration. In our work, we are satisfied with automatically preventing this misbehaviour for enough `invocations` by combining batching and multiple references. The use of both techniques yields good results for most of the UDFs studied and allows us to further optimize the translation. Compared to the PL/pgSQL version, only *ray, sheet* and *margin* are slightly slower. We get the biggest advantages with *bbox, march, march-tvf* and *vm* since we compute variables that depend on internal database tables, which thanks to our techniques are now processed using `Hash Joins`. Thereby, it can be said that our heuristic is nearly optimal.
It remains unclear whether we need to adapt the heuristic for larger and more complex UDFs, for example by working with multiple CTEs, each of which extends `run` by certain columns. This could be beneficial if we use multiple variables in a UDF, each with its own dependencies on additional tables. Instead of the recursive transformation as we use it here, a selective more general path extraction in the CFG could also be considered. However, it has to be investigated by which criteria this should be done. The performance gain through `WITH ITERATE` has already been mentioned. This feature should be provided by default and in the best case automatically be decided by the DBMS whether we need the behaviour of `WITH RECURSIVE` or `WITH ITERATE`.
Looking at DuckDB shows that batching in DBMSs with *vectorization* makes much more sense than with iterative behaviour through `Nested Loop Joins`. PostgreSQL itself has parallel operators after all, which could also be useful here, but they are only used by changing the internal parameters `parallel_setup_cost` and `parallel_tuple_cost`. However, the *parallel safety* restriction of PostgreSQL[1] prevents such operators from being used in temporary tables, i.e. also in working tables, so that the optimizations here can only be used for the initialization of the recursion. Unfortunately, this does not result in any improvements in our examples. Therefore, another possible approach could be to define or allow parallel operators for the recursive part of a CTE in PostgreSQL. From the use of DuckDB, we are forced to get new translations via case distinctions directly in the columns. Further research can be done in this direction as well, i.e. to investigate to what extent new translations are possible.
Regarding the internals of PostgreSQL, it is still possible to adapt *pgconfig* to one's hardware

[1] https://www.postgresql.org/docs/14/parallel-safety.html

and thus possibly accomplish more optimal plans. For example, `random_page_cost` are usually much too high for modern SSDs. Also, *JIT* is an important component, which we disabled contrary to the current default setting. Here it is interesting to investigate when the overhead caused by *JIT* in the planning can be recovered during runtime. For example, for *sched, savings* and *late*, one can see that for small parameters the overhead is significantly larger than we have optimization potential within the short runtime.
All our previous and possible future results should then be implemented in the already existing compiler. One should always take into account how long the compilation times are and deduce from this, in which cases this effort can be worthwhile. In any case, this work can contribute to the fact that the old developer wisdom *"Avoid PL/SQL if you can..."* will one day be obsolete and the writing of queries by using imperative techniques will be efficient enough for those programmers, who have no deep SQL knowledge.

Bibliography

[DHG19] Christian Duta, Denis Hirn, and Torsten Grust, *Compiling pl/sql away*, 2019.

[DKG18] Bailu Ding, Lucja Kot, and Johannes Gehrke, *Improving optimistic concurrency control through transaction batching and operation reordering*, Proc. VLDB Endow. **12** (2018), no. 2, 169–182.

[HG20] Denis Hirn and Torsten Grust, *Pl/sql without the pl*, Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '20, Association for Computing Machinery, 2020, p. 2677–2680.

[HG21] Denis Hirn and Torsten Grust, *One with recursive is worth many gotos*, p. 723–735, Association for Computing Machinery, New York, NY, USA, 2021.

[Leo20] Anghel Leonard, *Batching*, pp. 297–355, Apress, Berkeley, CA, 2020.

[McC76] T.J. McCabe, *A complexity measure*, IEEE Transactions on Software Engineering **SE-2** (1976), no. 4, 308–320.

[MSD93] M. Mehta, V. Soloviev, and D.J. DeWitt, *Batch scheduling in parallel database systems*, Proceedings of IEEE 9th International Conference on Data Engineering, 1993, pp. 400–410.

[PTH+17] Linnea Passing, Manuel Then, N. C. Hubig, Harald Lang, Michael Schreier, Stephan Günnemann, Alfons Kemper, and Thomas Neumann, *Sql- and operator-centric data analytics in relational main-memory databases*, EDBT, 2017.

[RP19] Karthik Ramachandra and Kwanghyun Park, *Blackmagic: Automatic inlining of scalar udfs into sql queries with froid*, Proc. VLDB Endow. **12** (2019), no. 12, 1810–1813.

[RPE+17] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham, *Froid: Optimization of imperative programs in a relational database*, Proc. VLDB Endow. **11** (2017), no. 4, 432–444.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Datum, Ort Unterschrift