Data is Data and Control Should be Data, Too

Compiling Iterative Table-valued PL/SQL UDFs into Recursive SQL Code

Denis Hirn[1]

[1] *supervised by Torsten Grust*
University of Tübingen, Germany

Abstract
PL/SQL functions suffer from poor runtime performance due to the frequent context switches that occur between the PL/SQL interpreter and the SQL executor. This switching causes friction that can slow down UDF execution significantly. *Table-valued* UDFs incur the additional challenge of the efficient treatment of the sizable results they generate. In this paper, we generalize our PL/SQL UDF compilation strategy to also handle such table-valued UDFs. The generated SQL code carefully separates control flow from data flow at runtime. Compiled UDFs efficiently stream their table-valued results (as opposed to UDF variants that need to hold and copy intermediate states in array variables) and thus impose significantly less memory pressure.

1. Introduction

PL/SQL is a high-level procedural programming language that allows developers to write custom user-defined functions (UDFs), operators, and algorithms that are not supported by the built-in functions of the database system. PL/SQL enables a style of imperative programming—which is quite different from the declarative set-oriented SQL paradigm—but still provides immediate access to database-resident tables. The distinctive PL/SQL features are (1) destructive variable assignments, (2) statement sequences, (3) arbitrary control flow (e.g. in terms of `IF…ELSE`, `WHILE`, `EXIT`), and a (4) seamless integration of SQL queries and expressions.

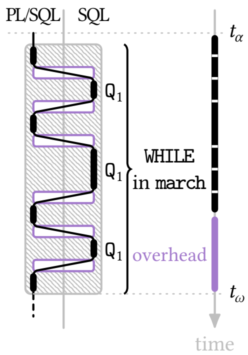

Figure 1: Context switching as UDF `march` executes.

Typically, PL/SQL is implemented as an interpreted language on top of SQL host engines. The imperative, non-SQL statements are interpreted by the PL/SQL subsystem, while all embedded SQL queries are sent to the SQL executor. Because both execution environments are completely disparate, each switch from one context to the other (and back) takes time and therefore causes context switching overhead. The situation is particularly dire when these embedded queries are placed in tight PL/SQL (`FOR` or `WHILE`) loops. In that case, switching back and forth between PL/SQL and the SQL executor occurs very frequently, which multiplies the overhead, and ultimately slows down execution. Figure 1 visualizes this.

It has thus become common developer lore that PL/SQL is slow and should be avoided if possible [1]. It has nonetheless been used for decades to implement complex database-driven applications [2]. Research has since recognized this as an important issue, resulting in several publications addressing this pressing challenge [1, 3–5].

The scope of the PhD project. The overarching goal of the PhD project is to allow developers to use imperative programming, e.g., in the form of PL/SQL UDFs, while maintaining the high performance of plan-based SQL execution. To accomplish this, we develop new ways to compile imperative PL/SQL UDFs into SQL queries. This compilation can handle arbitrary nesting of the PL/SQL features mentioned on the left. This includes looping control flow. As a side effect of this effort, database systems without PL/SQL support—but with support for a contemporary SQL dialect—will be able to run imperative PL/SQL UDFs after compilation, since no PL/SQL interpreter is required. In the present paper, we focus on the compilation of imperative (typically: iterative) table-valued UDF code into recursive yet plain SQL queries.

1.1. From Scalar Values to Tables

In [5], we described a compiler that transforms *scalar* PL/SQL UDFs to a single recursive SQL CTE (`WITH RECURSIVE`). While keeping the basic idea and compilation chain as is, the present work separates the management of control flow and data flow to make the compilation suitable for *table-valued* UDFs. To this end, we introduce the concept of *control rows* and *data rows*. Previously, the compiler used only control rows and could not handle table-valued UDFs.

Let us look at an example. UDF `march` of Figure 2a is a

VLDB 2023 PhD Workshop, co-located with the 49th International Conference on Very Large Data Bases (VLDB 2023), August 28, 2023, Vancouver, Canada
denis.hirn@uni-tuebingen.de (D. Hirn)
0000-0001-7040-1780 (D. Hirn)

```
CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
  DECLARE
    goal vec2 := start;
    cur  vec2 := start;
    dir  vec2;

  BEGIN
    WHILE true LOOP
      dir := (SELECT d.dir
              FROM   directions AS d, squares AS s
              WHERE  s.xy = cur
              AND    (s.ll, s.lr, s.ul, s.ur)
                     = (d.ll, d.lr, d.ul, d.ur));
      RETURN NEXT cur;
      cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
      EXIT WHEN cur = goal OR dir IS NULL;
    END LOOP;

  END;
$$ LANGUAGE PLPGSQL STRICT;
```

(a) **Table-Valued version of PL/SQL UDF** `march`**.**

```
CREATE FUNCTION march-arr(start vec2) RETURNS vec2[] AS $$
  DECLARE
    goal vec2 := start;
    cur  vec2 := start;
    dir  vec2;
    result vec2[] := ARRAY[] :: vec2[];
  BEGIN
    WHILE true LOOP
      dir := (SELECT d.dir
              FROM   directions AS d, squares AS s
              WHERE  s.xy = cur
              AND    (s.ll, s.lr, s.ul, s.ur)
                     = (d.ll, d.lr, d.ul, d.ur));

      result := result || cur;

      cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
      EXIT WHEN cur = goal OR dir IS NULL;
    END LOOP;
    RETURN result;
  END;
$$ LANGUAGE PLPGSQL STRICT;
```

(b) **Marching Squares as an array-based PL/SQL UDF.**

Figure 2: $Q_1[\cdot]$ is an embedded SQL query with the free variable `cur`.

table-valued function and implements the popular computer graphics algorithm *Marching Squares* [6] in PostgreSQL's PL/pgSQL. Whenever such a table-valued UDF encounters a `RETURN NEXT`, the PostgreSQL interpreter adds a new result to the function's result set before the UDF resumes execution. This, potentially sizable, return set is materialized during execution and returned as a whole when the function exits.

An alternative implementation as a *scalar* PL/SQL UDF (see `march-arr` in Figure 2b) iteratively builds the result as an array of type `vec2[]`. Our previous compilation strategy does handle `march-arr`, but the resulting SQL query will exhibit disappointing performance: the compilation creates a recursive CTE whose iteration expresses the iteration of the original UDF. This CTE maintains the local state of all UDF variables in a single row of the CTE's working table. For UDF `march-arr`, maintaining the array `result` iteratively results in significant runtime overhead because the array has to be copied (and extended) in each iteration. For n iterations, this amounts to a total of $n \times (1 + 2 + \ldots + (n-1)) \equiv \frac{1}{2}n^2 \times (n-1)$ copy operations. In consequence, the CTE's working table grows to 16 MB during the execution of the compiled UDF `march-arr`. Note that this copy overhead is not specific to PostgreSQL. It is a consequence of (semi-)naive evaluation of recursive CTEs. This evaluation semantics requires all data structures to be purely functional, meaning that they preserve previous versions of themselves unchanged. It may be possible to apply ideas from purely functional data structures [7] to database engines to improve the performance of recursive CTEs. However, as far as we know, no system currently does this.

In what follows, we sketch how to adapt and generalize the CTE-based PL/SQL UDF compilation to also cover table-valued UDFs like `march` of Figure 2a. We aim to (1) support the idiomatic `RETURN NEXT` style of UDF authoring, (2) avoid the materialization and copying of intermediate results, and (3) still avoid PL/SQL↔SQL switch overhead (which adds to 20% in the case of UDF `march`).

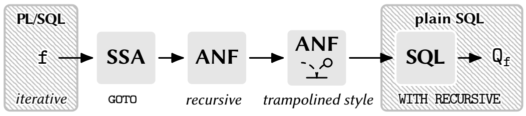

Figure 3: Compilation stages and intermediate UDF forms. See [5] for a detailed description of this compilation.

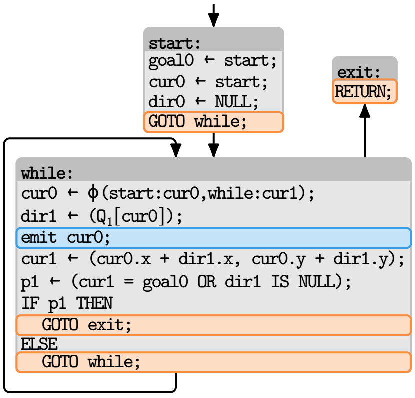

Figure 4: CFG for UDF `march` with code blocks in SSA form.

2. Trampolined Style in SQL

Complex UDFs that express (potentially deeply nested) looping computation and the simple iterative semantics of SQL's `WITH RECURSIVE` appear to be at odds. Yet, that gap can be bridged.

A program in trampolined style is organized as a single "scheduler" loop, the so called *trampoline*, which manages all control flow. Execution of such programs proceed in discrete steps. After each step, control is returned to the trampoline, which then proceeds to transfer control

```
1  WITH RECURSIVE run("rec?","data?",call,res,cur) AS (
2   SELECT true AS "rec?", false AS "data?", 'while' AS call, NULL::vec2 AS res, start AS cur
3     UNION ALL -- recursive UNION ALL
4   SELECT result.*
5   FROM   run,
6   LATERAL (SELECT if_p1.*
7            FROM     ( Q1[run.cur] ) AS let_dir(dir),
8            LATERAL (SELECT NULL AS "rec?", true AS "data?", NULL AS call, run.cur AS res, NULL AS cur
9                       UNION ALL
10                     SELECT if_p2.*
11                     FROM    (SELECT ((run.cur).x + dir.x, (run.cur).y + dir.y) :: vec2) AS let_cur(cur),
12                     LATERAL (SELECT let_cur.cur = start OR dir IS NULL) AS let_p1(p1),
13                     LATERAL (SELECT true AS "rec?", false AS "data?", 'while' AS call, NULL AS res, let_cur.cur AS cur     WHILE
14                              WHERE  NOT p1
15                                UNION ALL
16                              SELECT true AS "rec?", false AS "data?", 'exit' AS call, NULL AS res, NULL AS cur
17                              WHERE  p1) AS if_p2
18                    ) AS if_p1
19          WHERE run.call = 'while'
20            UNION ALL
21          SELECT false AS "rec?", false AS "data?", NULL AS call, NULL AS res, NULL AS cur     EXIT
22          WHERE run.call = 'exit') AS result
23   WHERE run."rec?")
24 SELECT run.res FROM run WHERE run."rec?" IS NULL AND run."data?";
```

Figure 5: Final plain SQL code emitted for the table-valued PL/SQL UDF `march` of Figure 2a.

again [8]. This cycle continues until the program execution is finished. Using trampolined style, the program is effectively transformed into a state machine. We exploit the property, that only a single loop is required to express arbitrary control flow. This restricted form of control flow perfectly matches the semantics of SQL's `WITH RECURSIVE` construct.

From PL/SQL to SQL. The compiler performs a series of transformations (see Figure 3) to get from PL/SQL to a plain recursive SQL query. The first step is to lower the UDF to *static single assignment* (SSA) form. Any iterative control flow is mapped to an equivalent program with labeled basic blocks that end with either `GOTO` κ to pass control to the block labeled κ, or `RETURN`. From the resulting CFG (see Figure 4) we derive the recursive SQL CTE `run` of Figure 5.

Control Flow Management. After compilation, each call to UDF `march` is encoded as a control row in the working table of `run`. This row determines the state of the machine, and thus which part of the computation to perform next. In Figure 4, each CFG construct that yields a control row is marked. It is initially created in the non-recursive part of `run` (see Line 2 of Figure 5). In the recursive part of `run`, the row is read, because two columns determine the transfer of control during execution:

`rec?` ∈ {`true, false`}: If column `rec?` is false, the trampoline will stop calculating and return.

`call` ∈ {`while`, `exit`}: Otherwise, column `call` specifies the block to jump to. When the block is finished, it returns a control row to the trampoline with new `rec?` and `call` values.

We call these rows *control rows*. The recursive part of `run` in Lines 4 to 23 of Figure 5 implements a dispatcher. Figure 6 depicts the central role of the dispatcher "trampoline" and how it realizes the control flow for UDF `march`. The SQL query reads the `call` column to select one block ∈ {`while`, `exit`} for evaluation. All blocks must return a new control row with columns `"rec?"` and `call`, so the dispatcher knows how to proceed in the next iteration (see Lines 13, 16, and 21 of Figure 5). This process continues until a block returns a control row with column `"rec?"=false` (see Line 21 of Figure 5). The working table in the next iteration will be empty, and `WITH RECURSIVE` evaluation stops.

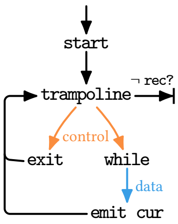

Figure 6: Trampolined style.

New: Data Flow Management. While scalar UDFs return a single value in the last trampoline iteration, table-valued UDFs can return any number of values during execution (see `emit cur0` in Figure 4). The CTE of Figure 5 encodes these returned values in dedicated rows marked in Line 8 of Figure 5. Two columns manage this data flow:

`data?` ∈ {`true, false`}: Column `data?` indicates if this row has a valid return value in column `res`.

`res`: Contains this return value.

We call rows with column `data?=true` *data rows*. When a UDF uses either `RETURN NEXT` or `RETURN QUERY`, such data rows are created in addition to control rows.

Given the UDF of Figure 2a and assuming a call `march((8,7))`, overall the recursive CTE computes table `run` as shown on the next page. After the initialization, marked ◆, each iteration (separated by – –) generates two rows, a data row and a control row. (In general, any number of data rows can be created in each iteration.) Note how the last iteration indicates the end of execution via `(rec?,data?)=(false,false)`.

run				
rec?	data?	call	res	cur
true	false	while	NULL	(8,7)
false	true	NULL	(8,7)	NULL
true	false	while	NULL	(9,7)
false	true	NULL	(9,7)	NULL
false	true	NULL	(8,8)	NULL
true	false	exit	NULL	(8,8)
false	false	NULL	NULL	NULL

Recall that the original PL/SQL UDF has to materialize its table-valued result during execution, and returns all of it as a whole. This materialization prevents the surrounding execution plan from terminating prematurely, for example, when a `LIMIT` clause is used: `SELECT * FROM march((8,7)) LIMIT 5`. After compilation, however, these results are immediately returned to the parent operator in terms of data rows, without materialization of the entire result. This saves memory and reduces the runtime. In addition, metrics such as CPU cost and cardinalities can be estimated more accurately, making planning of the translation more effective: While PL/SQL UDFs are effectively a black box for the planner, the translation is a regular SQL query that the planner is designed to handle.

3. Data Rows in Trampolined Style

Both UDFs, `march` and `march-arr`, indeed exhibit the infamous context switching overhead that gives PL/SQL its bad reputation. We have measured that the back and forth between PL/SQL and SQL accounts for 20% of the overall evaluation time for both variants (see Table 1). The compilation to recursive SQL CTEs described in [5] avoids this particular overhead for the two UDFs.

However, the naive treatment of the iterative `result` array construction and copying in the CTE for the scalar UDF `march-arr` quickly eats up all the gains: the quadratic array maintenance cost mentioned in the introduction add up to about 50% of the overall CTE runtime. If we double the size of UDF input, the working table of the CTE for `march-arr` grows by a factor of four (from 16 MB to 64 MB) and the array maintenance overhead increases to 56%. Ultimately, this leads to a *slowdown* of `march-arr` after compilation.

In stark contrast, the control- and data-flow-aware compilation strategy sketched in Section 2, translates the table-valued UDF into the recursive SQL CTE of Figure 5. Array construction and copying is avoided altogether and working table size remains small: doubling the UDF input size—and thus the number of iterations performed—linearly grows the working table's size from 110 kB to a mere 220 kB. Overall, compilation of UDF march leads to runtime reduction of 62% (i.e., post-compilation the UDF runs about 2.6 times faster). In addition, materialization is entirely avoided: the CTE of Figure 5 can stream the rows of the resulting table to the downstream plan.

Table 1
The context switching overhead before and speedup as well as working table size after compilation.

UDF	Return Type	Overhead	Runtime	Memory
march-arr	vec2[]	20%	112.8% (0.88×)	16 MB
march	SETOF vec2	20%	38.2% (2.61×)	110 kB

4. Wrap-Up

Separating the concepts of data rows and control rows is essential for translating table-valued functions. We save working table space and are rewarded with a significant runtime advantage over array-centric UDF alternatives. But it does not stop there. In the future, we plan to generalize this concept to add support for recursion in UDFs. Data rows could be used to model call stack entries, which—in one form or another—are required for functions that are not tail-recursive.

A further generalization would be to remove the restriction that one control row always yields exactly one new control row. This property causes trampolined style to model single-threaded computation. If the SQL backend supports parallel plan execution, the creation of multiple control rows in a single iteration effectively spawns independent threads. UDF evaluation would benefit from parallelization just like regular SQL queries.

References

[1] K. Ramachandra, K. Park, K. Emani, A. Halverson, C. Galindo-Legaria, C. Cunningham, Froid: Optimization of Imperative Programs in a Relational Database, Proc. VLDB 11 (2018).

[2] J. Harris, A (Not So) Brief But (Very) Accurate History of PL/SQL, 2020. http://oracle-internals.com/blog/2020/04/29/a-not-so-brief-but-very-accurate-history-of-pl-sql/.

[3] V. Simhadri, K. Ramachandra, A. Chaitanya, R. Guravannavar, S. Sudarshan, Decorrelation of user defined function invocations in queries, in: 2014 IEEE 30th ICDE, 2014.

[4] S. Gupta, S. Purandare, K. Ramachandra, Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates, in: Proc. SIGMOD, 2020.

[5] D. Hirn, T. Grust, One `WITH RECURSIVE` is Worth Many `GOTO`s, in: Proc. SIGMOD, 2021.

[6] C. Maple, Geometric design and space planning using the marching squares and marching cube algorithms, in: Proc. GMAG 2003, 2003.

[7] C. Okasaki, Purely Functional Data Structures, Cambridge University Press, USA, 1998.

[8] S. E. Ganz, D. P. Friedman, M. Wand, Trampolined style, in: Proceedings of the fourth ACM SIGPLAN ICFP, 1999.