Compiling PL/SQL Away

Christian Duta Denis Hirn Torsten Grust
Universität Tübingen
Tübingen, Germany
[christian.duta, denis.hirn, torsten.grust]@uni-tuebingen.de

ABSTRACT

"PL/SQL functions are slow," is common developer wisdom that derives from the tension between set-oriented SQL evaluation and statement-by-statement PL/SQL interpretation. We pursue the radical approach of compiling PL/SQL away, turning interpreted functions into regular subqueries that can then be efficiently evaluated together with their embracing SQL query, avoiding any PL/SQL ↔ SQL context switches. Input PL/SQL functions may exhibit arbitrary control flow. Iteration, in particular, is compiled into SQL-level recursion. RDBMSs across the board reward this compilation effort with significant run time savings that render established developer lore questionable.

1. NOW IS NOT A GOOD TIME TO INTERRUPT ME

Frequent changes | The required | of context | context switching effort | can turn | may even outweigh | otherwise tractable tasks | the cost | into real challenges. | of the tasks themselves.

If you have found those two sentences hard to comprehend, you were struggling with the *context switches*—occurring at every | bar—needed to process a piece of one sentence before immediately turning focus back to the other.

SQL evaluation in relational DBMSs can face such frequent context switches, in particular if bits of the query logic are implemented using PL/SQL,[1] *the* in-database scripting language. Whenever a SQL query `Q` invokes a PL/SQL function, say `f`,

- the DBMS switches from set-oriented plan evaluation to statement-by-statement PL/SQL interpretation mode (referred to as switch `Q→f` in the sequel).
- Execution of `f`'s statements then switches query evaluation back to plan mode—possibly multiple times—to evaluate the SQL queries $\mathtt{Q}_i$ embedded in `f` (switch $\mathtt{f{\to}Q}_i$).

Context switches will be abundant. If `f`'s call site is located inside a `SELECT-FROM-WHERE` block of `Q`, each row processed by the block will invoke `f`. Likewise, if `f` embeds multiple queries or employs iteration, *e.g.*, in terms of `FOR` or `WHILE` loops, we observe repeated plan evaluation for the $\mathtt{Q}_i$.

Unfortunately, both kinds of context switches are costly. Each switch `Q→f` incurs overhead for PL/SQL interpreter invocation or resumption. A switch $\mathtt{f{\to}Q}_i$ leads to overhead due to (1) plan generation and caching on the first evaluation of $\mathtt{Q}_i$ or (2) plan cache lookup, plan instantiation, and plan teardown for each subsequent evaluation of $\mathtt{Q}_i$. Iteration in both, `Q` and `f`, multiplies the toll.

Let us make the conundrum concrete with PL/pgSQL function `walk()` of Figure 3. The function simulates the walk of a robot 🤖 on a grid whose cells hold rewards (see Figures 1a and 2a). On cell (x, y) the robot follows a prescribed policy (*e.g.*, move down ↓ if on cell $(3, 0)$, see Figures 1b and 2b). This policy has been precomputed by a Markov decision process which takes into account that the robot may stray from its prescribed path: a planned move right from $(3, 2)$ will reach $(4, 2)$ with probability 80% but may actually end up in $(3, 3)$ or $(3, 2)$, each with probability 10% (see Figures 1c and 2c). A call $\mathtt{walk}(o,w,l,s)$ starts the robot in origin cell o and performs a maximum of s steps; `walk` returns early if the accumulated reward exceeds w or falls below l.

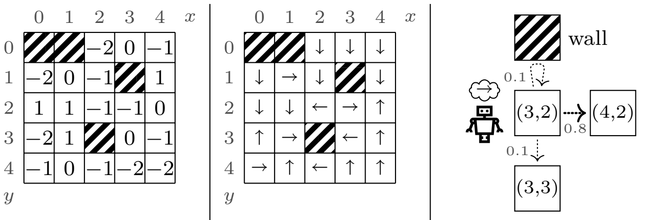
Each execution of PL/SQL function `walk` leads to the iterated evaluation of the embedded SQL queries $\mathtt{Q}_{1\ldots3}$. The run time profile on the rightmost edge of Figure 3 identifies these embedded queries to use the lion share of execution time (*e.g.*, $\mathtt{Q}_2$ accounts for 54.02% of `walk`'s overall run time). While we expect such embedded queries to dominate over the evaluation of simpler expressions and statements, the profile also shows that a significant portion of the evaluation time for the $\mathtt{Q}_i$ stems from $\mathtt{walk{\to}Q}_i$ context switch overhead (see the black section ▬ of the profile bars). For PostgreSQL, this cost is to be attributed to the engine's `ExecutorStart` and `ExecutorEnd` functions. These prepare the $\mathtt{Q}_i$'s plans (*i.e.*, copy the cached plan into a runtime data structure and instantiate the query's placeholders) and free temporary memory contexts, respectively. The `FOR` loop iteration in `walk` multiplies this effort. The bottom line shows that PostgreSQL invests more than 35% of its time in $\mathtt{walk{\to}Q}_i$ overhead during each invocation of `walk`. Section 3 shows similar or worse bad news for more PL/pgSQL functions.

Two worlds of interpreters. We stress that the roots of this tension between SQL and PL/SQL lie deep. We do not merely observe a deficiency of the languages' implementation in a specific RDBMS, say, PostgreSQL.

---

[1] We refer to the language as *PL/SQL* as coined by Oracle. Our discussion extends to its variants known as *PL/pgSQL* in PostgreSQL or *T-SQL* in Microsoft SQL Server.

10th Annual Conference on Innovative Data Systems Research (CIDR '20)
January 12–15, 2020, Amsterdam, Netherlands.



(a) Cell rewards. **(b) Markov policy.** **(c) Random straying.**

Figure 1: Controlling an unreliable robot to collect rewards.

cells

loc	reward
(2,0)	-2
(3,0)	0
(4,0)	-1
(0,1)	-2
⋮	⋮
(4,4)	-2

(a) Rewards.

policy

loc	action
(2,0)	↓
(3,0)	↓
(4,0)	↓
(0,1)	↓
⋮	⋮
(4,4)	↑

(b) Policy.

actions

here	action	there	prob
⋮	⋮	⋮	⋮
(3,2)	→	(4,2)	0.8
(3,2)	→	(3,3)	0.1
(3,2)	→	(3,2)	0.1
⋮	⋮	⋮	⋮

(c) Actions and straying.

Figure 2: A tabular encoding of the robot control scenario.

Once a (pure) SQL query has been translated into an internal tree of algebraic operators, its evalution is driven by a very specifically tuned plan interpreter: (1) a limited set of operators of known interface and behavior are orchestrated such that operator fusion or transitions between row-by-row and batched evaluation are feasible. Further, (2) the interpreter realizes a rigid evaluation discipline—in Volcano-style, for example—following predetermined paths of control. The deliberate control flow inside an PL/SQL function calls for a different imperative-style of interpreter whose progress is determined by the then current state of updateable variables. The function body is assembled from arbitrary blocks of statements whose behavior (*"will it loop, will it exit early?"*) is not known a priori.

A merger of the SQL and PL/SQL interpreters thus appears to be elusive. We regard the friction between both and the resulting context switch costs to be fundamental. (The situation may be different in DBMSs that *compile* SQL and PL/SQL into a common intermediate form that is then evaluated by a single interpreter or even the CPU if the IR is native to the host machine. The present work is concerned with *interpreted* query evaluation.)

Froid. PL/SQL has long been identified as a culprit for disappointing database application performance and it is common developer wisdom to "avoid PL/SQL functions altogether if possible" [11]. The situation is dire and has led to recent drastic efforts—coined *Froid* [11]—by the Microsoft SQL Server team: if function `f` **is simple enough**, compile its statements into a regular SQL subquery $\mathtt{Q_f}$ that can be inlined into the containing SQL query `Q`. Queries `Q` and $\mathtt{Q_f}$ may then be planned once and executed together in set-oriented fashion, avoiding any `Q→f` or $\mathtt{f{\to}Q}_i$ overheads. SQL Server with *Froid* indeed enjoys noticeable performance improvements and has been recognized as a major step forward by both, the developer as well as the database research communities [9].

In a nutshell, *Froid* transforms sequences of PL/SQL assignment statements into subqueries that are chained together with SQL Server's `OUTER APPLY` [6, 11]. The technique is elegant and simple but comes with severe restrictions: foremost, the just mentioned chaining will only work for functions `f` that exhibit loop-less control flow. This rules out PL/SQL functions like `walk` that build on `WHILE` or `FOR` iteration, arguably core constructs in any imperative programming language.

Compile PL/SQL away. We, too, subscribe to the drastic approach of *Froid*. However, we also believe that efforts that aim to host complex computation inside the DBMS and thus close to the data, need to support expressive programming language dialects. Control flow restrictions will be an immediate show stopper for the majority of interesting computational workloads. The present research thus sets out

1. to completely compile PL/SQL functions `f` away, transforming them into regular SQL queries $\mathtt{Q_f}$. The PL/SQL functions may feature iteration—in fact any control flow is acceptable. If `f` indeed contained iteration, $\mathtt{Q_f}$ will employ a recursive common table expression (CTE, `WITH RECURSIVE`) to express this in pure SQL. No changes to the underlying DBMS are required (although modest local changes can provide another boost, see Section 3).
2. We study and quantify the run time impact of this compilation approach and the benefit of getting rid of `Q→f` and $\mathtt{f{\to}Q}_i$ context switches, in particular.
3. As a by-product, the approach enables in-database programming support for DBMSs like SQLite3 that previously lacked any PL/SQL support at all.

Section 2, the core of this paper, elaborates on the compilation technique that turns iterative PL/SQL into recursive SQL. We hope to show that the transformation is systematic and practical. Along the way, we point out several opportunities to make the approach even more efficient. Section 3 reports on experimental observations we made once we compiled PL/SQL away.

2. COMPILING PL/SQL AWAY

The following structures the compilation into a series of transformation steps. We will use the PL/SQL function `walk` of Figure 3 as a running example and show interim results after each step. These intermediate forms of `walk` reveal further optimizations and simplifications we could apply underway. The four forms are (also see Figure 4):

- **SSA** Turn PL/SQL function `f` into *static single assignment* (SSA) form. This maps the diversity of PL/SQL control flow constructs to the single `GOTO` primitive.
- **ANF** From the SSA form derive a functional *administrative normal form* (ANF) for `f` which expresses iteration in terms of (mutually tail-)recursive functions.
- **UDF** Flatten mutual recursion and map the ANF functions into one tail-recursive SQL user-defined function.
- **SQL** Identify recursive calls and base cases in the body of this UDF and embed the body into a template query based on `WITH RECURSIVE`. This yields the SQL query $\mathtt{Q_f}$ we are after.

Query $\mathtt{Q_f}$ may then be inlined into `Q` at the call sites of the original function `f`.

SSA Explicit Data Flow and Simple `GOTO`-Based Control Flow

Lowering the PL/SQL input into its *static single assignment* (SSA) form [1] preserves the function body's impera-

```
-- move robot following a precomputed Markov policy
CREATE FUNCTION walk(origin coord, win int, loose int, steps int)
RETURNS int AS $$
DECLARE
 reward   int   = 0;
 location coord = origin;
 movement text  = '';
 roll     float;
BEGIN
 -- move robot repeatedly
 FOR step IN 1..steps LOOP
  -- where does the Markov policy send the robot from here?
  movement = (SELECT p.action
              FROM   policy AS p
              WHERE  location = p.loc);         Q1[·]
  -- compute new location of robot,
  -- robot may randomly stray from policy's direction
  roll = random();
  location =
    (SELECT move.loc
     FROM   (SELECT a.there AS loc,
                    COALESCE(SUM(a.prob) OVER lt, 0.0) AS lo,
                             SUM(a.prob) OVER leq      AS hi
             FROM   actions AS a
             WHERE  location = a.here AND movement = a.action
             WINDOW leq AS (ORDER BY a.there),
                    lt  AS (leq ROWS UNBOUNDED PRECEDING
                                EXCLUDE CURRENT ROW)
            ) AS move(loc, lo, hi)
     WHERE  roll BETWEEN move.lo AND move.hi);         Q2[·,·,·]
  -- robot collects reward (or penalty) at new location
  reward = reward + (SELECT c.reward
                     FROM   cells AS c
                     WHERE  location = c.loc);    Q3[·]
  -- bail out if we win or loose early
  IF reward >= win OR reward <= loose THEN
    RETURN step * sign(reward);
  END IF;
 END LOOP;
 -- draw: robot performed all steps without winning or losing
 RETURN 0;
END;
$$ LANGUAGE PLPGSQL;
```

Figure 3: Original PL/pgSQL function `walk`. Black sections of the profile bars ▬ quantify f→Q$_i$ context switch overhead.

tive style but introduces the invariant that any variable is now assigned exactly once (see Figure 5). Variable reassignment in the original function leads to the introduction of a new variable version (*e.g.*, $\mathtt{step}_2$ in Line 15) in SSA form. ϕ functions model that an assignment might be reached via different control flow paths. The SSA invariant facilitates a wide range of code simplifications, among these the tracking of redundant code, constant propagation, or strength reduction. Others have studied these in depth [5]. Let us note that PL/SQL code is subject to the same optimizations as any imperative programming language.

Statements in SSA programs are deliberately simple, featuring assignments, conditionals, `GOTO`s, and `RETURN` only. In the PL/SQL case, expressions in these SSA programs are regular SQL expressions. The SSA program contains the original `walk`'s embedded queries $\mathtt{Q}_{1\ldots3}$, with their query parameters instantiated by the appropriate SSA variables (see $\mathtt{Q}_1[\mathtt{location}_1]$ in Line 11, for example).

Importantly, the zoo of PL/SQL control flow constructs—including `LOOP`, `EXIT` (to label), `CONTINUE` (at label), `FOREACH`, `FOR`, `WHILE`— are now exclusively expressed in terms of `GOTO` and jump labels Lx. While verbose (the original `FOR` loop is now implemented by the conditional `GOTO` in Lines 8 to 10, the assignment of Line 15, and the `GOTO L1` of Line 18, for example), this homogeneity aids subsequent steps that trade

```
 1 FUNCTION walk(origin, win, loose, steps)
 2 {
 3   L0: GOTO L1;
 4   L1: reward_1   ← φ(L0: 0, L2: reward_2);
 5       location_1 ← φ(L0: origin, L2: location_2);
 6       movement_1 ← φ(L0: '', L2: movement_2);
 7       step_1     ← φ(L0: 0, L2: step_2);

 8       IF step_1 <= steps THEN
 9         GOTO L2
10       ELSE RETURN 0;

11   L2: movement_2 ← (Q_1[location_1]);
12       roll ← random();
13       location_2 ← (Q_2[location_1, movement_2, roll]);
14       reward_2 ← (Q_3[location_2]);
15       step_2 ← step_1 + 1;

16       IF reward_2 >= win OR reward_2 <= loose THEN
17         RETURN step_1 * sign(reward_2);
18       GOTO L1;
19 }
```

Figure 5: Iterative SSA form of PL/SQL function `walk`.

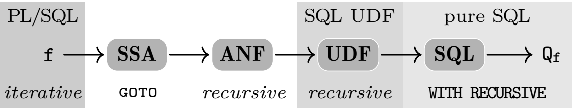


Figure 4: Intermediate forms on the way from f to Q$_f$.

```
1  FUNCTION walk(origin, win, loose, steps) =
2    LETREC L1(reward1, location1, movement1, step1) =
3      LETREC L2(reward1, location1, movement1, step1) =
4        LET movement2 = (Q1[location1])
5        IN
6          LET roll = random()
7          IN
8            LET location2 = (Q2[location1,movement2,roll])
9            IN
10             LET reward2 = reward1 + Q3[location2]
11             IN
12               LET step2 = step1 + 1
13               IN
14                 IF reward2 >= win OR reward2 <= loose THEN
15                   step1 * sign(reward2)
16                 ELSE
17                   L1(reward2,location2,movement2,step2)
18     IN
19       IF step1 <= steps THEN
20         L2(reward1,location1,movement1,step1)
21       ELSE 0
22   IN
23     L1(0, origin, '', 0)
```

Figure 6: Tail-recursive ANF variant of function `walk`.

control flow for function calls.

ANF Turning Iteration Into Tail Recursion

Despite its imperative appearance, the single-assignment restriction renders SSA already quite close to the functional *administrative normal form* (ANF) [2]. To translate from SSA to ANF we adapt an algorithm by Chakravarty and colleagues [3]. The resulting programs are purely expression-based and are composed of—besides basic subexpressions which, as for SSA, directly follow SQL syntax and semantics—`LET(REC)·IN` and `IF·THEN·ELSE` expressions only.

Put briefly, we arrive at the ANF of function `walk` (shown in Figure 6) by

- translating each jump label Lx and the statement block it governs into a separate function Lx(),
- turning `GOTO` Lx into calls to function Lx(), while
- supplying the values of ϕ-bound variables in Lx() as parameters to these function calls (we additionally perform lambda lifting and supply free variables as explicit parameters).

If we follow this strategy, iteration (*i.e.*, looping back to a label) will turn into recursion. Any such recursive call will be in *tail position* (since control does not return after a `GOTO`; see the calls to `L1()` in Lines 17 and 23 and `L2()` in Line 20) which will be crucial in the final translation to SQL's `WITH RECURSIVE`.

Finally, note how sequences of statements have turned into chains of nested `LET`s which nicely prepares the transcription to a SQL UDF in the upcoming step.

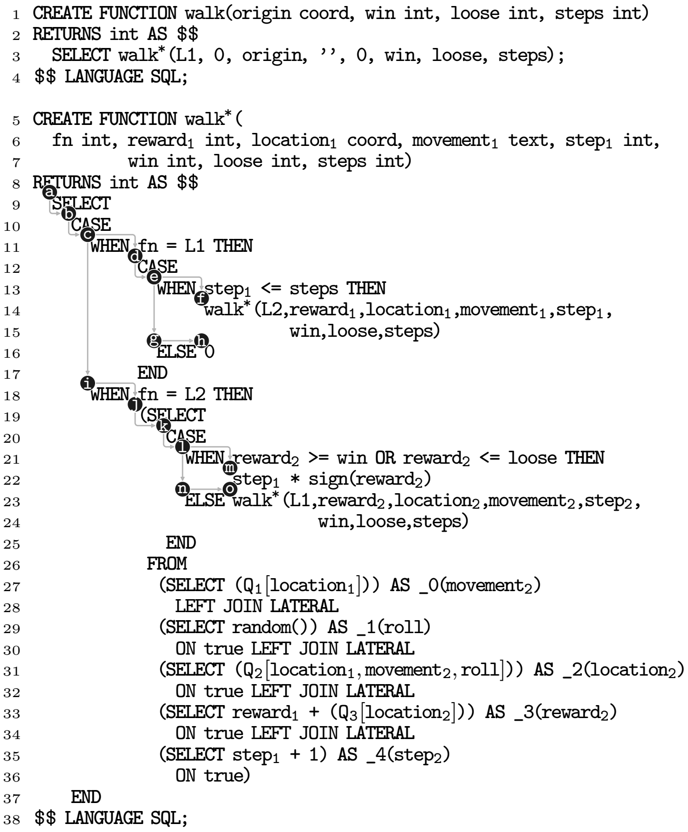
UDF Direct Tail Recursion in a SQL UDF

We take a first step towards SQL and transcribe the intermediate ANF into a user-defined SQL function (UDF). See Figure 7 (ignore the ● annotations for now). The mutual recursion between functions `L1()` and `L2()` is flattened using an additional parameter `fn` whose value discerns between the two call targets. This conversion into direct recursion follows standard defunctionalization tactics [7, 12], but inlining would work as well.

We follow *Froid* and compile ANF constructs `LET·IN` and `IF·THEN·ELSE` into SQL's table-less `SELECT` and `CASE·WHEN`, respectively. Nested `LET` bindings translate into `SELECT`s that are chained using `LEFT JOIN LATERAL`. If $[\![e]\!]$ denotes the SQL equivalent of ANF expression e, we have

$$[\![\texttt{LET}\ v = e_1\ \texttt{IN}\ e_2]\!] = \texttt{SELECT}\ [\![e_1]\!]\ \texttt{AS \_}(v)\ \texttt{LEFT JOIN LATERAL}\ [\![e_2]\!]\ \texttt{ON true}\ .$$

`LATERAL`, introduced by the SQL:1999 standard, implements the dependency of e_2 on variable v. In a sense, `LATERAL` thus assumes the role of statement sequencing via `;` in PL/SQL. Here, *Froid* relied on the Microsoft SQL Server-specific `OUTER APPLY` instead [6, 11]. The resulting `LATERAL` chains may look intimidating but note that these joins process *single-row tables* containing bindings of names to (scalar) values.



```
1  CREATE FUNCTION walk(origin coord, win int, loose int, steps int)
2  RETURNS int AS $$
3    SELECT walk*(L1, 0, origin, '', 0, win, loose, steps);
4  $$ LANGUAGE SQL;

5  CREATE FUNCTION walk*(
6    fn int, reward1 int, location1 coord, movement1 text, step1 int,
7            win int, loose int, steps int)
8  RETURNS int AS $$
9    SELECT
10     CASE
11       WHEN fn = L1 THEN
12         CASE
13           WHEN step1 <= steps THEN
14             walk*(L2,reward1,location1,movement1,step1,
15                        win,loose,steps)
16           ELSE 0
17         END
18       WHEN fn = L2 THEN
19         (SELECT
20            CASE
21              WHEN reward2 >= win OR reward2 <= loose THEN
22                step1 * sign(reward2)
23              ELSE walk*(L1,reward2,location2,movement2,step2,
24                            win,loose,steps)
25            END
26          FROM
27            (SELECT (Q1[location1])) AS _0(movement2)
28              LEFT JOIN LATERAL
29            (SELECT random()) AS _1(roll)
30              ON true LEFT JOIN LATERAL
31            (SELECT (Q2[location1,movement2,roll])) AS _2(location2)
32              ON true LEFT JOIN LATERAL
33            (SELECT reward1 + (Q3[location2])) AS _3(reward2)
34              ON true LEFT JOIN LATERAL
35            (SELECT step1 + 1) AS _4(step2)
36              ON true)
37     END
38 $$ LANGUAGE SQL;
```

Figure 7: Recursive SQL UDF `walk*` and its wrapper `walk`. The overlaid AST (●—●) becomes relevant in step SQL.

This translation step emits a regular SQL UDF which features direct tail recursion—in the case of `walk*` of Figure 7 we find two recursive call sites at Lines 14 and 23. DBMSs that admit such recursive UDFs could, in principle, evaluate this function to compute the result of the original PL/SQL procedure. We observe, however, that

- some DBMSs—among these MySQL, for example—forbid recursion in user-defined SQL functions, and that
- the direct evaluation of these UDF has disappointing performance characteristics. This is, again, due to significant `Q→f` and `f→Q` overhead: the plan for UDF `f*`'s body needs to be prepared and instantiated anew on each recursive invocation. Additionally, we quickly hit default stack depth limits, *e.g.*, in PostgreSQL or SQL Server.

```
1   WITH RECURSIVE run("call?", a̅r̅g̅s̅, result) AS (
2     -- original function call
3     SELECT true AS "call?", i̅n̅ AS a̅r̅g̅s̅, CAST(NULL AS τ) AS result
4       UNION ALL
5     -- subsequent recursive calls and base cases
6     SELECT iter.*
7     FROM   run AS r,
8            LATERAL (body(f*,r)) AS iter("call?", a̅r̅g̅s̅, result)
9     WHERE  r."call?"
10  )
11  -- extract result of final recursive function invocation
12  SELECT r.result
13  FROM   run AS r
14  WHERE  NOT r."call?"
```

Figure 8: SQL CTE template that evaluates tail-recursive $\texttt{f}^*$.

SQL Inlinable SQL CTE (`WITH RECURSIVE`)

Instead, we bank on a SQL:1999-style *recursive CTE* [13, § 7.12] as an evaluation strategy for recursion, ultimately compiling any use of PL/SQL or SQL user-defined functions away. The CTE constructs a table `run`$(\texttt{call?},\overline{\texttt{args}},\texttt{result})$[2] that tracks the evaluation of the recursive UDF $\texttt{f}^*$:

`call?`	Does the UDF perform a recursive call (`true`) or evaluate a base case (`false`)?
$\overline{\texttt{args}}$	In case of a call, what arguments are passed to $\texttt{f}^*$?
`result`	In a base case, what is the function's result?

Recall that we are dealing with tail recursion: once we reach a base case, the UDF's result is known and no further recursive ascent is required. The obtained result may thus be returned as the original function's outcome.

The evaluation of call $\texttt{f}(\overline{\texttt{in}})$ is expressed by the simple `WITH RECURSIVE` SQL code template of Figure 8:

- Line 3: Start evaluation with the original invocation of the UDF for argument list $\overline{\texttt{in}}$. $\texttt{f}^*$'s result (of type τ) is yet unknown and thus encoded as `NULL`.
- Line 9: Continue evaluation as long as new recursive calls are to be performed.
- Line 8: Evaluate the body of $\texttt{f}^*$ for the current arguments held in $\texttt{r.}\overline{\texttt{args}}$. This either leads to a new call or the evaluation of a base case.
- Lines 12 to 14: Once we reach a base, extract its result. Done.

The code template of Figure 8 is entirely generic. It is to be completed with a slightly adapted body—$\textit{body}(\texttt{f}^*,\texttt{r})$—of the UDF $\texttt{f}^*$ for `f`. In this adaptation,

- a recursive call $\texttt{f}^*(\overline{\textit{args}})$ is replaced by the construction of row $(\texttt{true},\overline{\textit{args}},\texttt{NULL})$ which encodes just that call in simulation table `run`,
- a base case expression with result v of type τ is replaced by row $(\texttt{false},\texttt{NULL},v)$.

Figure 9 depicts the resulting body $\textit{body}(\texttt{walk}^*,\texttt{r})$ for the recursive UDF of Figure 7. The construction of $\textit{body}(\texttt{f}^*,\cdot)$ calls for a simple abstract syntax tree (AST) traversal of the body of UDF $\texttt{f}^*$. Selected fragments of the AST for function $\texttt{walk}^*$ are shown in an overlay of Figure 7. This traversal identifies the leaves of the computation—*i.e.*, the recursive call sites ⓕ, ⓞ and base case expressions ⓗ, ⓜ—and performs the local replacements described above.

```
SELECT
  CASE
    WHEN r.fn = L1 THEN
         CASE
           WHENⓕr.step_1 <= r.steps THEN
               ROW(true,
                   ROW(L2,r.reward_1,r.location_1,r.movement_1,r.step_1,
                          r.win, r.loose, r.steps),
                   NULL)
           ELSEⓗROW(false, NULL, 0)
         END
    WHEN r.fn = L2 THEN
         (SELECT
            CASE
              WHENⓜreward_2 >= r.win OR reward_2 <= r.loose THEN
                  ROW(false,NULL,r.step_1 * sign(reward_2))
              ELSEⓞROW(true,
                      ROW(L1,reward_2,location_2,movement_2,step_2,
                             r.win, r.loose, r.steps),
                      NULL)
            END
          FROM
           (.  -- code of Figure 7
            :) -- (binds reward_2,location_2,movement_2,step_2)
  END
```

Figure 9: Adapted UDF body $\textit{body}(\texttt{walk}^*,\texttt{r})$. At ⓕ, ⓞ, and ⓗ, ⓜ row construction replaces recursive calls and base cases.

Finalization. A merge of $\textit{body}(\texttt{f}^*,\texttt{r})$ with the SQL code template yields a pure SQL expression which may be inlined at `f`'s call sites in the embracing query `Q`. Any occurrence of PL/SQL has been compiled away. The DBMS will be able to compile the resulting SQL query into a regular plan and jointly optimize the formerly separated code of `Q`, the transformed body of `f`, and the embedded queries $\texttt{Q}_i$. Most importantly, the evaluation of `Q` instantiates this joint plan *once* and will proceed without the need for `Q`→`f` or `f`→$\texttt{Q}_i$ context switches. The upcoming section quantifies the benefits we can now reap.

Beyond tail recursion. Let us close this discussion by noting that the `WITH RECURSIVE`-based simulation of a recursive function extends beyond tail recursion. Table `run` can be generalized to hold a true *call graph* that then does support recursive ascent. While this is not needed in the context of the present work, this paves the way for an intuitive, functional-style notation for SQL UDFs that may employ linear or general n-way recursion. The run time savings can be—again, due to the absence of plan instantiation effort—significant. We are actively pursuing this idea in a parallel strand of research that aims to leave more complex recursive computation in the hands of the DBMS itself.

3. ONCE PL/SQL IS GONE

Function `walk()` is not the exception. The described overheads are pervasive [11] and we, too, observed them across a variety of PL/SQL functions.

Context switching overhead. Table 1 contains a sample of iterative functions and reports the run time for repeated plan instantiation and deallocation to evaluate their embedded queries. Columns `Exec·Start` and `Exec·End` embody the `f`→$\texttt{Q}_i$ context switch overhead present in PostgreSQL (recall Section 1). Across the functions, we find overall `f`→$\texttt{Q}_i$ overheads of up to 38%. Only the columns `Exec·Run` and **Interp** represent productive evaluation effort: the execution of embedded queries and PL/SQL interpretation, respectively. Function `fibonacci`, an iterative computation of the nth Fibonacci number, evaluates arithmetic expressions only and

---

[2] $\overline{\texttt{args}}$ abbreviates the list of UDF arguments. For $\texttt{walk}^*$, $\overline{\texttt{args}} = \texttt{fn}, \texttt{reward}_1, \texttt{location}_1, \texttt{movement}_1, \texttt{step}_1, \texttt{win}, \texttt{loose}, \texttt{steps}$.

Table 1: Run time spent (in %) during PL/SQL evaluation. Bold entries indicate context switch overhead of kind `f`→`Q`$_i$.

PL/pgSQL Function	Exec·Start	Exec·Run	Exec·End	Interp
`walk` see Figure 3	**30.89**	55.13	**4.36**	9.63
`parse` via finite state automaton	**13.84**	68.52	**2.20**	15.62
`traverse` directed graph traversal	**31.80**	35.82	**6.03**	26.35
`fibonacci` iteratively compute *fib*(n)	**0**	90.45	**0**	9.55

does not execute embedded queries. PostgreSQL evaluates such *simple expressions* using a fast path that already foregoes plan instantiation. Compiling PL/SQL away does not promise much in this case. Still, turning query-less iterative functions into pure SQL can uncover opportunities for parallel evaluation—this is a direction we have not yet explored.

Iterative PL/SQL vs. Recursive SQL. For PL/pgSQL function `walk()`, Table 1 indicates potential run time savings of about 35% ≈ 30.89%+4.36% should we manage to get rid of context switching overhead. The translation from iterative PL/SQL to pure SQL built on a recursive CTE can indeed realize this advantage. Figure 10 shows the wall clock time of one invocation of `walk()` for a growing number of `FOR` loop iterations (which is controlled by parameter `steps`, see Figure 3). Throughout the experiment, the recursive SQL variant consistently shows an even greater run time savings of approximately 43%. Beyond saved context switches, this suggests that the evaluation of pure SQL expressions generally undercuts the interpretation of PL/SQL statements.

We have found the underlying RDBMSs to cope well with the resulting SQL queries and their associated plans. The SQL equivalent of function `walk()` accounts for a translation and optimization time of about 25 ms on PostgreSQL. As expected, the plans feature their share of `LATERAL` joins. Since these come with a prescribed order of evaluation (from left to right) and process single-row tables, the joins do not present a challenge to plan enumeration, however. We further observed that the sub-plans associated with the `Q`$_i$ before and after compilation did not diverge, essentially.

(The measurements of Figure 10 have been taken on a PostgreSQL 11.3 instance hosted on a Linux-based x86 box with 8 Intel Core$^{\text{TM}}$ i7 CPUs running at 3.66 GHz with 64 GB of RAM. We report the average as well as the window of minimal/maximal measurements of ten runs.)

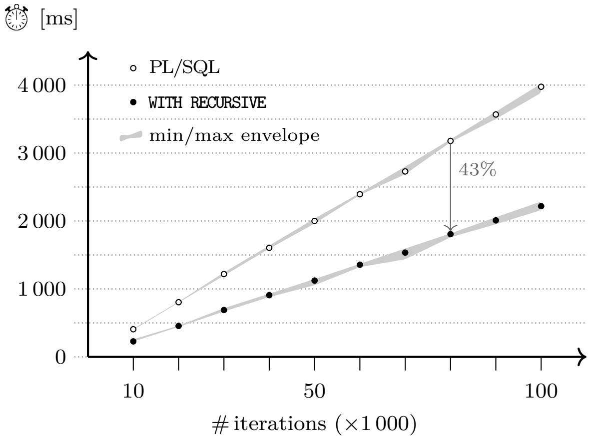


Figure 10: Iterative vs. recursive: wall clock time for `walk()` on PostgreSQL 11.3 across varying intra-function iterations.

Scaling the number of context switches. We can quite consistently observe these savings of ⩾ 40% across a wide range of scenarios. Figure 11a varies the number of invocations of `walk()` as well as the intra-function `FOR` loop iterations to obtain a heat map of run time improvements. Only very small numbers of invocations and/or iterations fail to compensate the one-time cost to optimize and evaluate the template query of Figure 8 (see the heat map's lower left). Beyond 32 invocations and/or iterations, the transformation to recursive SQL always is a clear win.

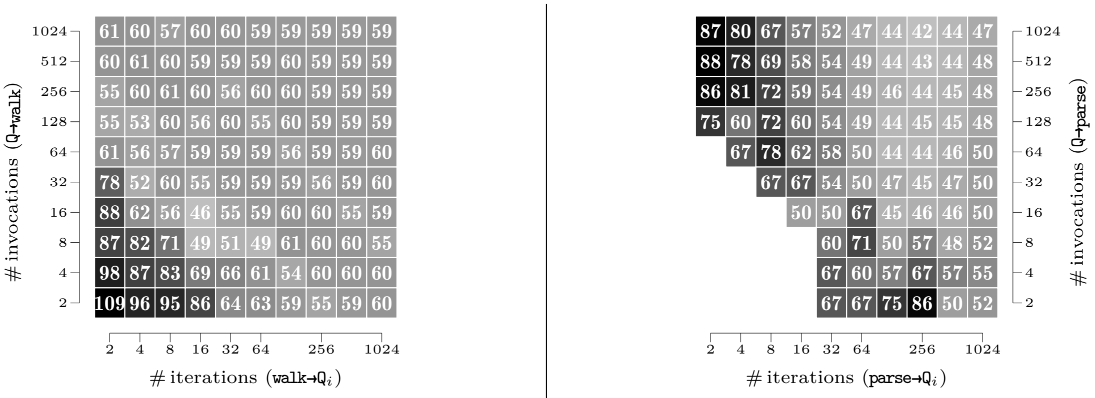
Beyond PostgreSQL. Modulo syntactic details, we were able to apply the function transformation of Section 2 immediately to Oracle, MySQL, SQL Server, and HyPer. As an example, Figure 11b shows how the evaluation of `parse()` on Oracle can significantly benefit once PL/SQL is traded for recursive SQL (measurements in the lower left appeared to be close to 100; we have omitted them here due to the DBMS's coarse timer resolution). SQLite3 lacks support for `LATERAL`, but a simple syntactic rewrite brought the functions to run on a system that formerly lacked any support for PL/SQL at all. Compiling PL/SQL away could, generally, pave the way to provide scripting support for more database engines.

When `WITH RECURSIVE` does too much. Exploiting tail recursion. The transformation from SSA to ANF compiles `GOTO` into *tail* recursion which obviates the need for recursive ascent: any activiation of a tail-recursive function already contains its complete evaluation context—typically held in the function's arguments. Tail recursion, thus, needs no stack (frames). Vanilla `WITH RECURSIVE`, however, collects a trace of all function invocations and their respective arguments (recall table `run` of Figure 8). For our purposes, accumulating this trace is wasted effort and the template of Figure 8 indeed uses the predicate `NOT r."call?"` in Line 14 to dispose of the trace and hold on to the function's *final* activation only.

Here, a hypothetical "`WITH TAIL RECURSIVE`" that keeps the most recent `run` row only would be a better fit. Interestingly, earlier work on the evaluation of complex analytics in HyPer has described just this construct, coined `WITH ITERATE` in [10]. To assess the benefit in the context of PL/SQL elimination, we implemented `WITH ITERATE` in PostgreSQL 11.3. The resulting space savings can indeed be profound, in particular for functions with potentially sizable arguments. One such example is function `parse()` which receives its input text as an argument. Table 2 lists the number of buffer page writes performed by PostgreSQL when inputs of growing length are parsed. `WITH ITERATE` realizes the promise of tail recursion and requires no space at all, while `WITH RECURSIVE` exhibits

Table 2: Eliminating buffering effort via `WITH ITERATE`.

# Iterations (= input length)	**# Buffer Page Writes** `WITH ITERATE`	 `WITH RECURSIVE`
10 000	0	6 132
20 000	0	24 471
30 000	0	55 016
40 000	0	97 769
50 000	0	152 729



# invocations (Q→walk)	2	4	8	16	32	64		256		1024
1024	61	60	57	60	60	59	59	59	59	59
512	60	61	60	59	59	59	60	59	59	59
256	55	60	61	60	56	60	60	59	59	59
128	55	53	60	56	60	55	60	59	59	59
64	61	56	57	59	59	59	56	59	59	60
32	78	52	60	55	59	59	59	56	59	60
16	88	62	56	46	55	59	60	60	55	59
8	87	82	71	49	51	49	61	60	60	55
4	98	87	83	69	66	61	54	60	60	60
2	109	96	95	86	64	63	59	55	59	60

(a) `walk` on PostgreSQL.

# invocations (Q→parse)	2	4	8	16	32	64		256		1024
1024	87	80	67	57	52	47	44	42	44	47
512	88	78	69	58	54	49	44	43	44	48
256	86	81	72	59	54	49	46	44	45	48
128	75	60	72	60	54	49	44	45	45	48
64		67	78	62	58	50	44	44	46	50
32			67	67	54	50	47	45	47	50
16				50	50	67	45	46	46	50
8					60	71	50	57	48	52
4					67	60	57	67	57	55
2					67	67	75	86	50	52

(b) `parse` on Oracle.

Figure 11: Relative run time (in %) of recursive SQL vs. iterative PL/SQL. Light colors indicate an advantage for SQL.

quadratic space appetite (both, the number of required iterations that consume one input character each as well as the lengths of the residual strings left to parse, do grow).

In an age of complex in-database computation, we step forward and propose that a construct like `WITH ITERATE` should find its way into the SQL standard.

4. (TOO EARLY FOR) CONCLUSIONS

This marks the beginning of a thread of research in which we aim to explore fresh ways to support complex in-database computation, preferably *without* turning existing engines on their head.

Current coverage of PL/SQL. The compilation strategy does not restrict the control flow used to express the imperative `f` and admits, for example, deep loop nesting (this is not showcased in the present paper). Exceptions and their associated handlers constitute more of a challenge in this respect: raising an exception is readily expressed in terms of SSA's `GOTO`, but the detection of exception conditions from within a SQL query appears difficult. PL/SQL variables of non-atomic types (*e.g.*, row values, arrays, or geometric objects) seamlessly fit with the compilation scheme as long as the underlying RDBMS supports their storage in table cells. A positional array update $\texttt{a[}i\texttt{]} = e$ as permitted by PL/SQL translates into a less efficient replacement of array `a`. We are currently underway to devise a compilation scheme for cursors that range over the result rows of an embedded query $\texttt{Q}_i$. Dynamic SQL (PL/SQL's string-based `EXECUTE`) will probably never compile to SQL.

Here, we have assumed the return type τ of PL/SQL function `f` to be scalar but this is not an inherent restriction. A generalization to set-returning functions (the `RETURN NEXT` of PL/SQL) has already been found to integrate elegantly.

Directions waiting to be explored include at least the following:

- With its recent Version 12, PostgreSQL will offer hooks that enable merging of CTEs with their containing queries. Inlining compiled functions into their calling query then opens up additional optimization opportunities.
- Flattening nested iteration into flat recursion facilitates efficient evaluation through partitioning and parallelism.
- If `f` is to be invoked $n > 1$ times (since it is embedded in a SQL query `Q`) *and* the arguments $\overline{\texttt{in}}$ of these calls are known beforehand, the proposed compilation scheme is able to perform all of these calls in terms of a single evaluation of the template of Figure 8: instead of the single-row recursive seed set up in Line 3 of the template, supply an n-row table of all arguments $\overline{\texttt{in}}$. All else remains as is. The query will return a table of n result rows without ever leaving the context of the `WITH RECURSIVE` block. Such *batching* [4, 8] has been identified to provide a substantial boost in the iterative evaluation of function calls and we should be able to benefit, too.
- Since the compilation discloses the formerly opaque internals of a function's body to the SQL query optimizer, we can expect a significantly better estimation of its evaluation cost (instead of the all too common default or fixed cost). Exactly how this improves plan quality for function-rich workloads remains to be quantified.
- Beyond PL/SQL: With its ability to compile arbitrary SSA programs, this provides the groundwork required for the compilation and evaluation of expressive imperative languages within regular DBMSs and thus close to the data.

5. REFERENCES

[1] B. Alpern, M. Wegman, and F. Zadeck. Detecting Equality of Values in Programs. In *Proc. POPL*, San Diego, CA, USA, 1988.

[2] A. Appel. SSA is Functional Programming. *ACM SIGPLAN Notices*, 33(4), 1998.

[3] M. Chakravarty, G. Keller, and P. Zadarnowski. A Functional Perspective on SSA Optimisation Algorithms. *Electronic Notes in Theoretical Computer Science*, 82(2), 2004.

[4] W. Cook and B. Wiedermann. Remote Batch Invocation for SQL Databases. In *Proc. DBPL*, Seattle, WA, USA, 2011.

[5] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM TOPLAS*, 13(4), 1991.

[6] C. Galindo-Legaria and M. Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *Proc. SIGMOD*, Santa Barbara, CA, USA, 2001.

[7] T. Grust, N. Schweinsberg, and A. Ulrich. Functions are Data Too (Defunctionalization for PL/SQL). In

Proc. VLDB, Riva del Garda, Italy, 2013.

[8] R. Guravannavar and S. Sudarshan. Rewriting Procedures for Batched Bindings. *Proc. VLDB*, 1(1), 2008.

[9] B. Ozar. Froid: How SQL Server 2019 Might Fix the Scalar Functions Problem. `www.brentozar.com`, 2018.

[10] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann. SQL- and Operator-Centric Data Analytics in Relational Main-Memory Databases. In *Proc. EDBT*, Venice, Italy, 2017.

[11] K. Ramachandra, K. Park, K. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB*, 11(4), 2018.

[12] J. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11, 1998.

[13] *SQL:1999. Database Languages–SQL–Part 2: Foundation*. ISO/IEC 9075-2:1999.