One WITH RECURSIVE is Worth Many GOTOs

Denis Hirn Torsten Grust
University of Tübingen
Tübingen, Germany
[denis.hirn,torsten.grust]@uni-tuebingen.de

ABSTRACT

PL/SQL integrates an imperative statement-by-statement style of programming with the plan-based evaluation of SQL queries. The disparity of both leads to friction at runtime, slowing PL/SQL execution down significantly. This work describes a compiler from PL/SQL UDFs to plain SQL queries. Post-compilation, evaluation entirely happens on the SQL side of the fence. With the friction gone, we observe execution times to improve by about a factor of 2, even for complex UDFs. The compiler builds on techniques long established by the programming language community. In particular, it uses trampolined style to compile arbitrarily nested iterative control flow in PL/SQL into SQL's recursive common table expressions.

1 "AVOID PL/SQL IF YOU CAN ..."

"Avoid PL/SQL if you can ..." is a paraphrased bit of developer wisdom that relates to the—often disappointing—runtime performance of user-defined functions (UDFs) in relational database systems [14, 37, 49]. This is particularly sobering since, despite the above advice, PL/SQL UDFs have been widely used as a backbone of complex database-backed applications for decades now [28].

As a procedural language extension to SQL, PL/SQL [43] has been designed to express complex computation right inside the DBMS, and thus close to the data it operates over [50]. PL/SQL deliberately admits an imperative coding style in which the majority of developers are already well versed. The key elements of PL/SQL are

- stateful variables of types that match the (scalar) types of SQL,
- blocks of statement sequences,
- arbitrarily complex and looping control flow (expressed in terms of, *e.g.*, IF...ELSE, LOOP, WHILE, FOR, EXIT, or CONTINUE), and a
- seamless embedding of simple SQL expressions and entire queries.

Such a tight integration of imperative programming with declarative SQL certainly bears promise for data-centric application development. It also reveals, however, a disparity between the statement-by-statement interpretation of a PL/SQL UDF f and the set-oriented plan-based evaluation of the SQL queries Q_i it embeds. We indeed find that the cost of switching between both worlds is a main cause of why PL/SQL execution is perceived as being "slow."

Let us make this concrete with the UDF route of Figure 2, implemented in PostgreSQL's variant PL/pgSQL of PL/SQL [46]. Given a network of point-to-point connections (see Figure 1a as an example), a call route(*s*,*d*,*ttl*) returns the array of intermediate nodes, or hops, that lie between source node *s* and destination *d*. To find this path, route consults table connections (Figure 1b) in which a row (h, t, v, c) indicates that the cheapest path from h to t goes via hop v for an overall cost of c. Should these costs exceed our budget *ttl*, route bails out and returns NULL. (connections is the *distance vector routing table* [23] for the network of Figure 1a.) Figure 1c shows a selection of source/dest pairs and the paths computed by route, assuming generous *ttl* budgets.

Observe how PL/SQL UDF route embeds several scalar SQL expressions like loc <> dest, ttl < hop.cost, or route || loc (see Lines 10, 14, and 18 in Figure 2). PostgreSQL's PL/SQL interpreter evaluates these simple SQL expressions via a *fast path* that directly invokes the system's SQL expression evaluator.

Notably, though, route also contains the SQL SELECT-block Q_1 which the function uses to query the routing table for the next hop from loc towards dest (the two PL/SQL variables loc and dest are free in the SELECT-block and act like parameters for Q_1, see Line 13 in Figure 2). While PostgreSQL compiles and optimizes Q_1 only once on its first encounter during the interpretation of route, *each of the potentially many evaluations* of the embedded query entails

(1) the instantiation of the runtime data structures for Q_1's plan (this includes copying in the current values of loc and dest),
(2) calling out to the SQL query engine to evaluate Q_1, and
(3) tearing down the plan's temporary structures, reclaiming memory, before communicating the result back to PL/SQL (here: binding variable hop in Line 11).

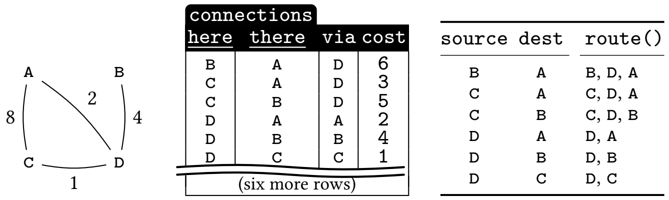
This constitutes the lion share of the context switching overhead between SQL and PL/SQL but still is only one part of the conundrum. To obtain the complete picture, we embed the invocation of PL/SQL UDF route in a top-level SQL query, say Q_0, that finds paths made up of more than two hops:

```
SELECT c.here, c.there, route(c.here, c.there, 10)
FROM   connections AS c                                  (Q0)
WHERE  cardinality(route(c.here, c.there, 10)) > 2;
```

While the plan for query Q_0 executes to scan the rows of table connections, we observe context switches in both directions:

Q→f [from (top-level) query Q to function f] Each evaluation of the WHERE and SELECT clauses in Q_0 invokes or resumes the PL/SQL interpreter to evaluate the embedded calls to UDF route.

f→Q [from function f to (embedded) query Q] During the interpretation of route, each execution of the assignment to hop in Line 11 of Figure 2 calls on the SQL engine to evaluate embedded query Q_1, which entails the plan preparation and cleanup



connections

here	there	via	cost
B	A	D	6
C	A	D	3
C	B	D	5
D	A	A	2
D	B	B	4
D	C	C	1
(six more rows)			

source	dest	route()
B	A	B, D, A
C	A	C, D, A
C	B	C, D, B
D	A	D, A
D	B	D, B
D	C	D, C

(a) Network. **(b) Routing table.** **(c) Sample routes.**

Figure 1: Sample network with distance-vector routing table.

```
1  CREATE FUNCTION route(source node, dest node, ttl int)
2  RETURNS node [] AS $$
3    DECLARE
4      route node[];                        -- path between source and dest
5      loc   node;                          -- current location in network
6      hop   connection;                    -- next hop leading from loc to dest
7    BEGIN
8      loc   := source;                     -- move to source
9      route := array[loc];                 -- path starts at source
10     WHILE loc <> dest LOOP               -- while dest has not been reached:
11       hop := (SELECT c          Q1[·,·]        -- consult routing table
12               FROM   connections AS c          --   to find next hop
13               WHERE  c.here = loc AND c.there = dest);
14       IF ttl < hop.cost THEN                   -- bail out if
15         RETURN NULL;                           --   budget exceeded
16       END IF;
17       loc   := hop.via;                        -- move to found hop
18       route := route || loc;                   -- add hop to path
19     END LOOP;
20     RETURN route;                        -- return constructed path
21   END;
22 $$ LANGUAGE PLPGSQL;
```

Figure 2: Original PL/SQL UDF `route`. $Q_1[\cdot,\cdot]$ denotes an embedded SQL query containing the free variables `loc` and `dest`.

steps (1) to (3) as discussed above. Importantly, since this assignment is placed inside a PL/SQL `WHILE` loop and thus will be iterated, one invocation of `route` may lead to several such `route`→Q_1 context switches.

Figure 3 depicts the resultant prevalent back and forth between PL/SQL and SQL. Read this timing diagram from top to bottom. We start in the SQL context (marked ▒) of top-level query Q_0 and switch to the PL/SQL interpreter on any invocation of UDF `route`. Interpretation of `route` is suspended—possibly many times due to `WHILE` loop iteration—to establish a new SQL context for embedded query Q_1. The execution of `RETURN` in Line 20 of `route` reinstates the top-level Q_0 context until the next `route` invocation occurs.

For all PL/SQLs UDFs in our experimental evaluation of Section 4, we counted on the order of 10^2 to 10^6 SQL↔PL/SQL border crossings during the evaluation of one top-level query, even if we disregard the relatively cheap *fast path* evaluations of simple scalar SQL expressions. These context switching efforts add up and significantly contribute to the overall execution time: in the time frame $t_\omega - t_\alpha$, only the time slices ▌ constitute useful work (*i.e.*, plan evaluation or statement interpretation). In PostgreSQL, where a Q→f switch is initiated by the system's `plpgsql_call_handler()` and its f→Q counterpart is implemented by the pair of kernel routines `ExecutorStart()` and `ExecutorEnd()` [45], we measured typical overheads between 10% and 60% of the overall runtime of the top-level query. Similar costs can be observed for other RDBMSs that pursue PL/SQL interpretation (ultimately, this led to the recent *Froid* [47, 49] effort in Microsoft SQL Server [39], for example).

While this spells bad news for interpreted PL/SQL, the language is undoubtedly pervasive, easily accessible for a wide range of developers due to its imperative flavor, and most probably not going away any time soon [7]. Can we keep writing PL/SQL UDFs but reduce the context switch burden?

1.1 Objective: Zero Context Switches

We pursue the goal of eliminating the context switches between SQL and PL/SQL and the associated expense altogether. To reach this ideal, we opt to evaluate both, the containing top-level query Q_0 as well as its embedded PL/SQL function(s) `f`, on the SQL side of the fence. **This calls for a translation of PL/SQL UDF `f` into a plain SQL query Q_f** that, when evaluated, yields the same result as a call to `f`. Inlining Q_f into Q_0 where the top-level query originally invoked `f` then yields a pure-SQL form of the computation. After inlining, the plan for $Q_0 + Q_f$ can be generated, optimized, instantiated, evaluated, and teared down *once*. During evaluation, no context switches to PL/SQL (due to `f`) occur and the RDBMS remains in its set-based plan evaluation mode throughout. Ideally, the pure-SQL form performs useful work only, yielding shorter query runtimes.

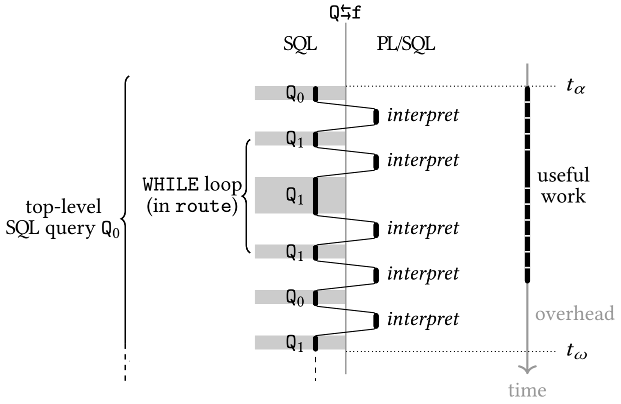


Figure 3: Context switching as top-level query Q_0 executes.

Contributions. The present work describes a systematic compilation process that derives a plain SQL:1999 query Q_f from a given PL/SQL UDF `f`, provided that `f` is a pure function that does not alter the underlying database.

- UDF `f` may exhibit **arbitrary conditional and looping control flow**, in particular (nested) iteration expressed in terms of PL/SQL's variety of `WHILE`, `FOR`, and `LOOP` constructs. Support for rich control flow clearly is critical to be able to formulate complex computation in PL/SQL. This renders the language Turing-complete and implies that the resulting compiled SQL query Q_f will need to bank on SQL:1999's *recursive common table expressions* (short: CTEs, `WITH RECURSIVE`) [19, 55] should `f` indeed contain loops. `WITH RECURSIVE` and its fixpoint semantics may appear alien to many developers but the present compiler will derive the required CTE automatically from given PL/SQL source. The compilation strategy is elaborated on in Section 2.
  The ability to compile arbitrary iterative control flow into plain SQL is a defining difference to earlier work, *Froid* [49] in particular. We enlarge the PL/SQL subset fit for compilation to SQL and thus considerably widen the reach of the approach.
- The compiler is realized as a source-to-source translation and does not invade the internals of the underlying RDBMS. In fact, any RDBMS with support for `WITH RECURSIVE` and `LATERAL` joins—both SQL language features that have been around for about two decades now [15, 55]—makes for a suitable host system. We mandate support for the mentioned SQL constructs but nothing beyond a contemporary SQL dialect is required. In particular, we can target RDBMSs that do not feature native processors for PL/SQL at all, *e.g.*, SQLite3 [56] or MySQL [40]: the

present work provides a foundation on which PL/SQL support for such systems could be built.
The technical Section 3 describes the compiler in sufficient detail such that readers should be able to fully re-implement it in their own system.

- UDF `route` of Figure 2 has been deliberately designed as a simple example that we will follow throughout the paper. A collection of further PL/SQL UDFs, from moderately to very complex, is in focus in Section 4. We assess the context switching overhead these functions induce on PostgreSQL 11.3 and do find that compilation to plain SQL can indeed recover this overhead and more (up to 60% runtime savings). The recursive evaluation of $\mathtt{Q}_\mathtt{f}$ may consume more memory than the interpretation of `f`. We adress this, too, in Section 4.

The source for all UDFs prior to and after compilation is available in a *GitHub* repository.[1]

2 ONE WAY TO TRADE PL/SQL FOR SQL

We organize the compiler as a series of stages, the first of which consumes the original PL/SQL UDF `f` and the last of which emits a plain SQL query $\mathtt{Q}_\mathtt{f}$ (see Figure 4). This section traverses the stages from front to back and gives a complete account of the compilation strategy. Section 3 fills in the technical details of how the intermediate program forms are transformed into each other and would be required reading if you plan to rebuild the compiler at your site.

Input to the compiler is a PL/SQL UDF that adheres to the grammar of Figure 5. The admissable subset of PL/SQL does not restrict the flow of control and, in particular, embraces various forms of loops which may be nested, cut short via `CONTINUE`, or left early via `EXIT` or `RETURN` [46, §43.6]. Non-terminal a represents embedded SQL expressions, *the* source of `f`→`Q` context switches during UDF interpretation. Here, we restrict UDFs to return values of scalar types τ to streamline the discussion. The compilation scheme naturally extends to tabular return types and, indeed, table-valued PL/SQL functions may have particular efficient SQL equivalents, see Section 4.1. The `STABLE` volatility annotation admits UDFs that may read but not alter the database state. This aids compilation into read-only SQL queries. (We currently explore a variety of ways to lift this restriction and admit updating UDFs, *e.g.*, in terms of *pending update lists* [54]. See Section 3.3.)

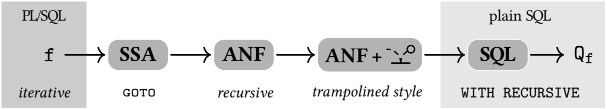


Figure 4: Compilation stages and intermediate UDF forms.

f ::=	`CREATE FUNCTION` v `(`$v\ \tau, \ldots, v\ \tau$`)` `RETURNS` τ `AS`
	`$$` $p$ `$$ LANGUAGE PLPGSQL STABLE;`
		PL/SQL UDF

$p$ ::=	`DECLARE` d `; BEGIN` s `; END`	UDF body
d ::=	$v\ \tau$ | $v\ \tau$ `:=` a | d `;` d	variable declarations
s ::=	v `:=` a	statements
	| `IF` a `THEN` s `; [ ELSE` s `; ] END IF`	
	| `LOOP` s `; END LOOP`	
	| `WHILE` a `LOOP` s `; END LOOP`	
	| `FOR` v `IN` a `..` a `LOOP` s `; END LOOP`	
	| `EXIT` | `CONTINUE` | `RETURN` a	
	| s `;` s	statement sequence
a ::=	`(`⟨*scalar SQL expression*⟩`)`	embedded SQL
v ::=	⟨*identifier*⟩	variable/function name
τ ::=	⟨*scalar SQL type*⟩	scalar value type

Figure 5: PL/SQL language subset covered by the compiler.

b ::=	κ `:` s `;`	labelled block
s ::=	$v \leftarrow a$	statement
	| `IF` v `THEN GOTO` κ `ELSE GOTO` κ	
	| `GOTO` κ	
	| `RETURN` a	
	| s `;` s	statement sequence
a ::=	`(`⟨*scalar SQL expression*⟩`)`	embedded SQL
v ::=	⟨*identifier*⟩	variable/procedure name
κ ::=	⟨*block label*⟩	jump target for `GOTO`

Figure 6: `GOTO`-based imperative intermediate form.

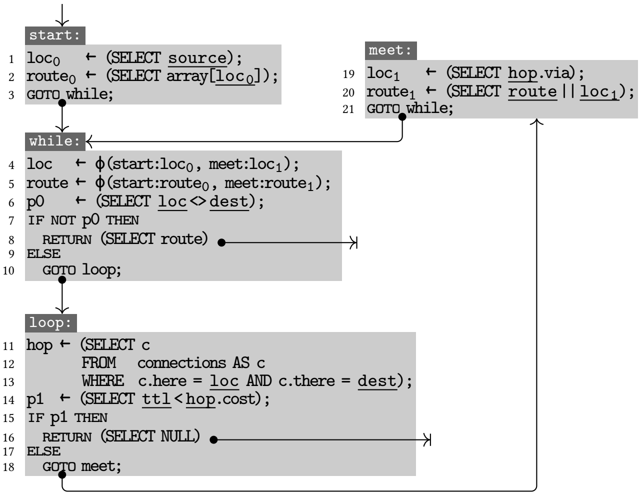
2.1 PL/SQL Control Flow in Terms of `GOTO`

The first stage aims to cut down language complexity and transforms the body of the input UDF into a deliberately simple imperative form (see the grammar of Figure 6). In this form, *all control flow* is expressed in terms of `IF`/`ELSE` and `GOTO`. Code is organized into labelled basic blocks which contain `;`-separated sequences of statements that either end in `GOTO` κ (to transfer control to the block with label κ) or `RETURN`. The simple `GOTO`-based form can capture arbitrary control flow patterns and streamlines the entire compilation pipeline. Section 3 details the translation from PL/SQL into the simplified imperative form in terms of mapping ⨟.

Let us use *control flow graphs* (CFGs) to visualize the resulting collection of basic blocks and the transfer of control between them. Figure 7 shows the CFG for the PL/SQL UDF `route` of Figure 2. Execution starts at the block labelled `start` and ends when `RETURN` hands control back to the UDF caller ($\bullet\!\!\rightarrow\!\!|$ edges). Embedded SQL expressions are preserved in `(`$\cdots$`)`, *i.e.*, the non-terminals a in the grammars of Figures 5 and 6 coincide. Loops in the PL/SQL UDF surface as cycles in the CFG form (*e.g.*, cycle $\texttt{while}\overset{\curvearrowleft}{\rightarrow}\texttt{loop}\rightarrow\texttt{meet}$ represents the `WHILE` loop in UDF `route`).

The CFG form facilitates a variety of program simplifications. *Block inlining* [9,18,58], in particular, reduces the number of `GOTO`s, an optimization that will turn out valuable (see Section 4). In the CFG of Figure 7, some inlining has already been performed, *e.g.*, two `GOTO` κ to blocks κ that contain `RETURN` a only have been replaced by that `RETURN` a (Lines 8 and 16). We bring the CFG into *static single assignment* form (SSA [12]) in which renaming ensures that any variable is assigned at a single program location (cf. the versions $\mathtt{loc}_{0,1}$ and $\mathtt{route}_{0,1}$ of variables `loc` and `route` in Figure 7). SSA paves the way for simplifications like dead code elimination or unused variable detection. Further, SSA's *phi functions* [12] make explicit which variable versions are live if a block can be entered from multiple predecessors, see ϕ`(start:·,meet:·)`

[1] PL/SQL and SQL source ready to run on PostgreSQL 11 (or newer) can be found at `http://github.com/One-WITH-RECURSIVE-is-Worth-Many-GOTOs`.



```
start:
1  loc_0   ← (SELECT source);
2  route_0 ← (SELECT array[loc_0]);
3  GOTO while;

while:
4  loc   ← ϕ(start:loc_0, meet:loc_1);
5  route ← ϕ(start:route_0, meet:route_1);
6  p0    ← (SELECT loc <> dest);
7  IF NOT p0 THEN
8    RETURN (SELECT route)
9  ELSE
10   GOTO loop;

loop:
11 hop ← (SELECT c
12        FROM   connections AS c
13        WHERE  c.here = loc AND c.there = dest);
14 p1  ← (SELECT ttl < hop.cost);
15 IF p1 THEN
16   RETURN (SELECT NULL)
17 ELSE
18   GOTO meet;

meet:
19 loc_1   ← (SELECT hop.via);
20 route_1 ← (SELECT route || loc_1);
21 GOTO while;
```

Figure 7: CFG for UDF `route` with code blocks in SSA form.

in Lines 4 and 5 of block `while`. This is vital information for the subsequent ANF conversion (Sections 2.2 and 3.1). Since we rely on well-established SSA ideas here, we opt to not elaborate.

However, let us not move on before noticing that the CFG form already is largely PL/SQL-agnostic: any imperative language that maps to the `GOTO`-based form of Figure 6 could use the CFG as a side entry and thus be compiled into plain, possibly recursive, SQL.

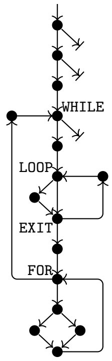


Figure 8: CFG of UDF `packing`.

The transformation of `route` led to a compact single-cycle CFG. UDFs that perform complex computational tasks will exhibit considerably more intricate control flow and yield CFGs of multiple cycles. One example is UDF `packing` which we will encounter in Section 4: its CFG, reproduced in Figure 8 on the left, contains three cycles. Among these are two `LOOP` and `FOR` loops nested in an enclosing `WHILE` iteration (a conditional `EXIT` ensures that the endless `LOOP` can be left eventually), leading to a loop nesting pattern of ⌴⌴. There is a tension between such complex loop constellations and the simple form of iteration (or recursion) that our compilation target, SQL:1999's `WITH RECURSIVE`, can provide [19, 55]. The following compilation stages cope with this challenge.

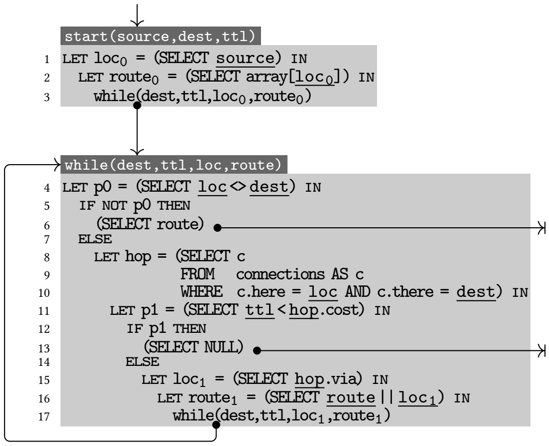
2.2 Tail Recursion Replaces GOTO

The second stage converts the imperative program in SSA form into a purely functional equivalent in *administrative normal form* (or ANF [51]). The equivalence of both forms is well-studied [1]—in Section 3.1, we describe one such conversion strategy that follows Chakravarty *et al.* [8]. We arrive at a deliberately simple functional language whose grammar is shown in Figure 9. Again, nonterminal a denotes the sub-language of embedded SQL expressions which the above conversion leaves intact.

In a nutshell, each basic block labelled κ turns into a function κ`()`. The block's sequence of assignment statements is expressed

$f ::= v\,(v\ \tau, \ldots, v\ \tau) : \tau = e$	function definition
$e ::= a$	expression
$\mid\ v\,(a, \ldots, a)$	
$\mid$ IF v THEN e ELSE e	
$\mid$ LET $v = a$ IN e	
$a ::= (\langle$*scalar SQL expression*$\rangle)$	embedded SQL
$v ::= \langle$*identifier*$\rangle$	variable/function name
$\tau ::= \langle$*scalar SQL type*$\rangle$	parameter/return type

Figure 9: Intermediate functional language in ANF.



```
start(source,dest,ttl)
LET loc_0 = (SELECT source) IN
  LET route_0 = (SELECT array[loc_0]) IN
    while(dest,ttl,loc_0,route_0)

while(dest,ttl,loc,route)
LET p0 = (SELECT loc <> dest) IN
  IF NOT p0 THEN
    (SELECT route)
  ELSE
    LET hop = (SELECT c
               FROM   connections AS c
               WHERE  c.here = loc AND c.there = dest) IN
      LET p1 = (SELECT ttl < hop.cost) IN
        IF p1 THEN
          (SELECT NULL)
        ELSE
          LET loc_1 = (SELECT hop.via) IN
            LET route_1 = (SELECT route || loc_1) IN
              while(dest,ttl,loc_1,route_1)
```

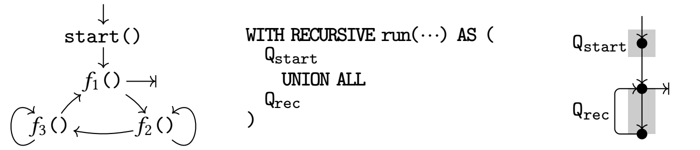
Figure 10: Call graph for UDF `route` after ANF conversion.

in terms of cascades of `LET` bindings. Like SSA, ANF binds variables to embedded SQL expressions a only. Conditionals translate straightforwardly. `GOTO` κ leads to a *tail call* to function κ`()`. We ultimately end up with a family of tail-recursive functions that mutually call each other. Figure 10 shows the resulting function *call graph* derived from the CFG of UDF `route` (Figure 7). We enter the computation by invoking function `start()` which then performs a tail call to `while()`. The latter recursively invokes itself until its two base cases eventually return the value of variable `route` or `NULL` to the caller.

Much like SSA, ANF is an intermediate program form that comes with a rich background of established optimizations [20,32]. Again, *inlining* is key to reduce the number of functions (nodes) and tail calls (edges) in the call graph. In Figure 10, in fact, functions `loop()` and `meet()` have already been inlined into `while()`, leaving us with a two-function family. Generally, exhaustive inlining leads to a call graph of $n + 1$ functions if the input CFG contains n cycles (+1 accounts for `start()`). We will later see that any tail call we can save is paid back in terms of SQL runtime reduction (Section 4).

2.3 Trampolined Style Tames Mutual Recursion

Since the call graph reflects the input UDF's iterative control flow, we may end up with quite intricate networks of mutually recursive functions, f_i`()` say. The graph has been simple for `route`, but for PL/SQL UDF `packing` and its three loops (recall Figure 8) we already arrive at the call graph of Figure 11a. Such complex function interaction is in stark contrast to the simple computational pattern that SQL:1999's `WITH RECURSIVE` provides [3, 55] (see Figure 11b):



```
WITH RECURSIVE run(···) AS (
  Qstart
    UNION ALL
  Qrec
)
```

(a) ANF call graph for UDF packing**.**

(b) A recursive CTE as defined in SQL:1999.

(c) Recursion in a CTE.

Figure 11: Call graph patterns: UDF *vs.* recursive CTE.

(1) Empty the *UNION table* which will hold the overall result. Evaluate SQL query $\mathtt{Q_{start}}$ and place its rows in a *working table.*
(2) If the working table is empty, return the UNION table as the final result. Otherwise, add the working table to the UNION table.
(3) Evaluate query $\mathtt{Q_{rec}}$ over the current working table (referred to as run in $\mathtt{Q_{rec}}$) and replace the working table with the resulting rows. Go to (2).

This simple linear recursion (or iteration) scheme corresponds with the call graph of Figure 11c. Only the simplest, single-loop single-exit UDFs would map straightforwardly to it.

To bridge that gap, we adopt *trampolined style* [22]. A program in trampolined style is organized as a single loop which repeatedly executes a *dispatcher* function (or: *trampoline*). Two arguments to the dispatcher control its operation:

rec? ∈ {true, false}: If rec? (read: *recurse further?*) is false, computation is complete. Return result.

call ∈ $\{f_1, \ldots, f_n\}$: Otherwise, invoke the function $f_i()$ selected by call. When f_i is done, it performs a tail call to the dispatcher, passing new rec? and call to indicate how computation shall continue.

If we apply trampolined style to the PL/SQL UDF packing and its call graph (Figure 11a), we obtain the single-cycle call graph reproduced in Figure 12. All $f_i()$ pass back control to the dispatcher upon function return, such that trampoline() is in full control of whether and how the computation proceeds.

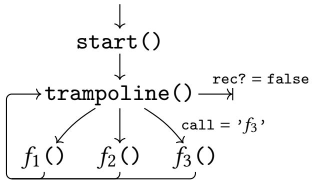
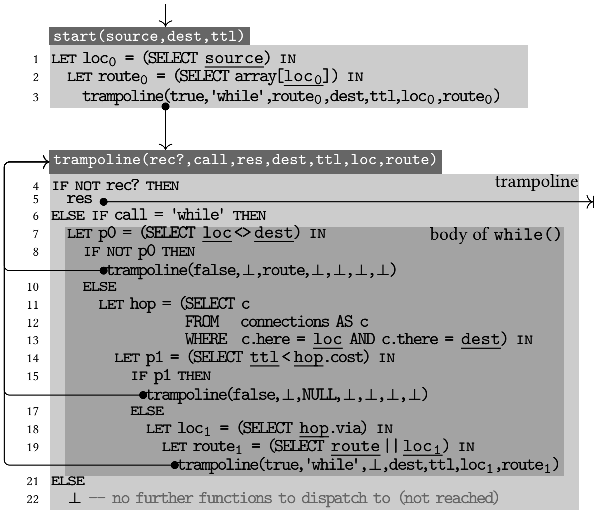


Figure 12: Trampolined style.

To zoom in on the details, we have rephrased the ANF program for UDF route of Figure 10 in trampolined style. The result is shown in Figure 13. The dispatcher trampoline() now is central and provides the only exit branch should argument rec? equal false (Line 5). This particular instance of a dispatcher can invoke one mere function (*i.e.*, we have call ∈ {while}) which is due to UDF route's simple control flow. Note that we have inlined the body of function while() into the dispatcher itself (Lines 7 to 20), saving calls from the dispatcher to while(). Where the original while() function returned to its caller, it now tail calls trampoline() with argument rec? = false and passes the appropriate res return value (variable route or NULL) in Lines 9 and 16. Where while() originally recursively invoked itself, it now passes rec? = true, call = 'while' to trampoline() to achieve the same effect (Line 20).



```
   start(source,dest,ttl)
 1 LET loc₀ = (SELECT source) IN
 2   LET route₀ = (SELECT array[loc₀]) IN
 3     trampoline(true,'while',route₀,dest,ttl,loc₀,route₀)

   trampoline(rec?,call,res,dest,ttl,loc,route)
 4 IF NOT rec? THEN                                        trampoline
 5   res
 6 ELSE IF call = 'while' THEN
 7   LET p0 = (SELECT loc<>dest) IN              body of while()
 8     IF NOT p0 THEN
         trampoline(false,⊥,route,⊥,⊥,⊥,⊥)
10     ELSE
11       LET hop = (SELECT c
12                  FROM   connections AS c
13                  WHERE  c.here = loc AND c.there = dest) IN
14         LET p1 = (SELECT ttl<hop.cost) IN
15           IF p1 THEN
               trampoline(false,⊥,NULL,⊥,⊥,⊥,⊥)
17           ELSE
18             LET loc₁ = (SELECT hop.via) IN
19               LET route₁ = (SELECT route||loc₁) IN
                   trampoline(true,'while',⊥,dest,ttl,loc₁,route₁)
21 ELSE
22   ⊥ -- no further functions to dispatch to (not reached)
```

Figure 13: Program in trampolined style (ANF body of route inlined into trampoline). Compare to Figure 10.

It is most important, however, that the call graph shapes of Figure 12 and Figure 13—and, in fact, *any* program in trampolined style—*do match* the restricted single-cycle iteration scheme implemented by WITH RECURSIVE (recall Figure 11c). Since arbitrary programs of mutually recursive functions are susceptible to the trampoline treatment [11], we build on a SQL formulation of trampolining in the last compiler stage.

2.4 Trampolined Style in SQL

This last stage transitions from functions in ANF to plain SQL. To this end, we will

(1) translate the function bodies of start(), $f_1(), \ldots, f_n()$ into $1 + n$ SELECT blocks $b_0, b_1, \ldots, b_n$, and then
(2) fit these blocks into a WITH RECURSIVE-based SQL template that implements the trampoline. The instantiated template constitutes the compiler's final output.

Step (1) consults each of the $1 + n$ bodies in turn to construct $1 + n$ independent SELECT (or SELECT-FROM) blocks. The body of $f_i()$ (and start()) comprises of cascaded LET expressions, IF-ELSE conditionals, and tail calls to trampoline(). (cf. Lines 7 to 20 in Figure 13). The translation of such non-looping, yet branching programs into plain SQL has been studied by *Froid* and its precursors [49, 53]. We largely adopt their strategy, generalizing it in the process (*e.g.*, through the use of the standard LATERAL instead of the Microsoft SQL Server-specific CROSS APPLY [21] to express the dependencies introduced by nested LETs). In Section 3, this translation from ANF functions to SQL is realized in terms of mapping ⤇.

When the resulting SELECT block b_i is evaluated, it returns a single row in which the leading columns "rec?", call, res (function result) inform the SQL-based dispatcher on how to proceed.[2] Trailing columns carry the arguments to the functions $f_i()$ as determined by ANF conversion (see dest, ttl, loc, route in Figures 10 and 13).

[2] A single row suffices to hold the function's result of scalar return type τ, see Figure 5. This naturally extends to b_i returning one or more rows in case we compile table-valued UDFs. We comment on this in Section 4.1.

```
1  WITH RECURSIVE run("rec?",call,res,...) AS (
2    SELECT true AS "rec?", 'f_i' AS call, ...                b_0 (start())
3      UNION ALL -- recursive UNION ALL
4    SELECT  result.*
5    FROM    run,
6    LATERAL (⟨b_1⟩ WHERE run.call = 'f_1'                    dispatcher
7               UNION ALL
8                   ⋮
9               UNION ALL
10             ⟨b_n⟩ WHERE run.call = 'f_n'
11           ) AS result
12   WHERE   run."rec?"
13 )
14 SELECT run.res FROM run WHERE NOT run."rec?";
```

Figure 14: SQL:1999-based trampoline template.

Step (2) uses the SELECT blocks $b_0, b_1, \ldots, b_n$ to plug the holes in the SQL-based trampoline template of Figure 14. You will find that this recursive CTE first evaluates the SELECT block b_0 (derived from ANF function `start()`, Line 2) before it repeatedly enters the dispatcher in Lines 6 to 11. The dispatcher inspects column `call` to select one b_i for evaluation (in this UNION ALL stack, only the b_i selected by `call` places its result row in the working table; Section 3 comments on this particular implemantion of dispatch and the translation of multi-way IF-ELSE conditionals in general). Since all b_i return columns `"rec?"` and `call`, the dispatcher will know how to proceed on the next iteration.

Once a b_i emits a row with column `"rec?"` = `false`, the working table of the next iteration will be empty (Line 12), prompting WITH RECURSIVE to stop. This final row and all result rows collected so far are returned in UNION table `run` (see Section 2.3 and [19]). Column `res` of the final row holds the overall result of the compiled UDF, which is exactly what we extract in Line 14.

Template instantiation (performed by mapping $\Mapsto$ described in Section 3) completes the compilation process. The instantiated SQL trampoline for the UDF `route` of Figure 2 is reproduced in Figure 15. We invite you to match this SQL code with `route`'s ANF form (Figure 10) and the SQL trampoline template of Figure 14. (To make that match perfect, note that variable `route` has been found to unify with `res` in this particular example and thus has been optimized away in Figure 15.)

Wherever the original database application invokes the PL/SQL `route(source,dest,ttl)`, the plain SQL code of Figure 15 can replace that call. Given the scenario of Figure 1 and assuming a call `route(C,A,10)`, the recursive CTE will compute the UNION table `run` shown on the left. Note that UDF table `run` carries a complete stack-like history of the computation: function `while()` has been entered three times (and returned the partial paths (C), (C,D), (C,D,A), respectively), before the final result path C,D,A (in the highlighted bottom `res` cell) can be extracted. Not a single context switch to PL/SQL has been performed to compute this result—in fact, the underlying RDBMS need not supply a PL/SQL interpreter at all.

run

rec?	call	res	loc
true	while	C	C
true	while	C,D	D
true	while	C,D,A	A
false	NULL	C,D,A	NULL

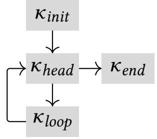
3 FROM PL/SQL TO SQL: BEHIND THE SCENES

Three syntactic transformations jointly implement the PL/SQL UDF compiler (recall Figure 4). We provide a detailed account of these three transformations in this section in terms of syntax-to-syntax mappings $\fatsemi$, $\Mapsto$, and $\mathrel{|\!\!\Mapsto}$. On the way, we invoke a SSA to ANF conversion as described by Chakravarty *et al.* [8] and also include notes on this step. *Inference rules* $\frac{\textit{antecedents}}{\textit{consequence}}$ define these mappings by syntactic cases: the case (or consequence) below the line follows if the antecedents above the line are satisfied.

3.1 From PL/SQL to SSA and ANF ($\fatsemi$)

$\fatsemi$ (and its two auxiliaries $\mapsto$, $\looparrowright$, all defined in Figure 16) maps the body of a PL/SQL UDF into a dictionary s of blocks (*i.e.*, statement sequences) of the simple imperative form defined in Figure 6. Within s, blocks are identified by their labels κ. The core is $\Gamma \vdash (c, s_1) \mapsto_{\kappa_1} (\kappa_2, s_2)$ which transforms the single PL/SQL instruction c into a sequence of simple imperative statements, all of which are appended to the statements in the block labelled κ_1. This updates the old label-to-block dictionary s_1 to s_2. (The antecedents perform this update via $s_2 \equiv s_1 +_{\kappa_1} [\langle \textit{statements} \rangle]$ which creates block κ_1 in s_2 if it does not already exist in s_1). Once c has been translated, subsequent statements are to be appended to block κ_2. $\fatsemi$ uses the continuation block label κ_2 to compile entire sequences of PL/SQL instructions (see Rules SEQ and SEQ0 in Figure 16).

When it comes to the compilation of iteration, LOOP, WHILE, and FOR are handled alike. $\mapsto$ calls on the auxiliary $\looparrowright$ (Rule ITER) which additionally returns a quadruple of labels $(\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end})$. Rules LOOP, WHILE, and FOR use the associated blocks to establish looping control flow as shown here on the right. This block arrangement helps to compile PL/SQL's CONTINUE and EXIT simply in terms of the imperative statements GOTO κ_{head} and GOTO κ_{end}, respectively (Rules CONT and EXIT). Since PL/SQL loops may nest, all rules pass on or maintain a stack Γ of $\langle \kappa_{head}, \kappa_{end} \rangle$ pairs of which the topmost entry relates to the currently innermost loop.



Given input PL/SQL UDF `f`(*params*) with statement sequence *cs* as its body and return type τ, we invoke $() \vdash (cs, \varnothing) \fatsemi_{\texttt{start}} (\kappa, s)$ to compile *cs* into a collection of blocks s (here, () and $\varnothing$ denote an empty stack and empty block dictionary, respectively; we can safely ignore the final continuation label κ). The blocks in the resulting s define the nodes, the GOTOs in these blocks define the edges of the CFG for UDF `f`. Execution begins in the unique block labelled `start` (cf. Section 2.1). The resulting CFG is not yet in SSA form, but it can be straightforwardly converted following standard algorithms [6, 12].

The control flow graph that connects a UDF's SSA blocks will generally be cyclic. Still, the CFG's *dominator tree* [12] organizes the blocks in a hierarchy: block κ_1 *dominates* κ_2 ($\kappa_1 \rightarrow \kappa_2$) if all control flow paths that reach κ_2 must pass through κ_1. Figure 17a shows the dominator tree for UDF `route`, reflecting its control flow graph of Figure 7.

From this dominator tree, SSA to ANF conversion [8] derives a hierarchy of nested functions (see Figure 17b). A block labelled κ_1 turns into a function κ_1`()` that encloses function κ_2`()` if $\kappa_1 \rightarrow \kappa_2$. An SSA inter-block jump GOTO κ turns into a tail call to κ`()` in ANF. Cycles in the control flow graph thus turn into loops in the function call graph.

```
WITH RECURSIVE run("rec?",call,res,loc) AS (
  SELECT true AS "rec?", 'while' AS call, array[source] AS res, source AS loc              } start()
    UNION ALL -- recursive UNION ALL
  SELECT  result.*                                                                          ⎫
  FROM    run,                                                                              ⎪
  LATERAL (SELECT  if_p0.*                                                                  ⎪
           FROM    (SELECT loc <> dest) AS let_p0(p0),                           LET p0     ⎪
           LATERAL (SELECT  false AS "rec?", NULL AS call, res, NULL AS loc  IF NOT p0      ⎪
                    WHERE   NOT p0                                                          ⎪
                      UNION ALL                                                             ⎪
                    SELECT  if_p1.*                                              ELSE (p0)  ⎪
                    FROM    (SELECT (SELECT c                                    LET hop    ⎬ while()
                                     FROM connection AS c                                   ⎪
                                     WHERE c.here = loc AND c.there = dest)) AS let_hop(hop),
                    LATERAL (SELECT ttl < (hop).cost) AS let_p1(p1),             LET p1     ⎪
                    LATERAL (SELECT false AS "rec?", NULL AS call, NULL AS res, NULL AS loc  IF p1
                             WHERE  p1                                                      ⎪
                               UNION ALL                                     ELSE (NOT p1)  ⎪
                             SELECT true AS "rec?", 'while' AS call, res || (hop).via AS res, (hop).via AS loc
                             WHERE  NOT p1) AS if_p1                                        ⎪
                    WHERE   p0) AS if_p0                                                    ⎪
           WHERE run.call = 'while') AS result                                              ⎭
  WHERE run."rec?"
)
SELECT run.res FROM run WHERE NOT run."rec?";
```

Figure 15: Final plain SQL code emitted for PL/SQL UDF `route` of Figure 2, complete with instantiated trampoline.

$$\frac{\Gamma \vdash (stmt, s) \mapsto_\kappa (\kappa_1, s_1) \qquad \Gamma \vdash (stmts, s_1) \fatsemi_{\kappa_1} (\kappa_2, s_2)}{\Gamma \vdash (stmt; stmts, s) \fatsemi_\kappa (\kappa_2, s_2)}(\textsc{Seq}) \qquad \frac{}{\Gamma \vdash (\varepsilon, s) \fatsemi_\kappa (\kappa, s)}(\textsc{Seq0})$$

$$\frac{\Gamma \vdash (vars, s) \fatsemi_\kappa (\kappa_1, s_1) \qquad \Gamma \vdash (stmts, s_1) \fatsemi_{\kappa_1} (\kappa_2, s_2)}{\Gamma \vdash (\texttt{DECLARE}\ vars\ \texttt{BEGIN}\ stmts\ \texttt{END}, s) \mapsto_\kappa (\kappa_2, s_2)}(\textsc{Body}) \qquad \frac{s_1 \equiv s +_\kappa [v \leftarrow q\,;]}{\Gamma \vdash (v\ \tau := q, s) \mapsto_\kappa (\kappa, s_1)}(\textsc{Decl}) \qquad \frac{s_1 \equiv s +_\kappa [v \leftarrow \texttt{(SELECT NULL)}\,;]}{\Gamma \vdash (v\ \tau, s) \mapsto_\kappa (\kappa, s_1)}(\textsc{Decl0})$$

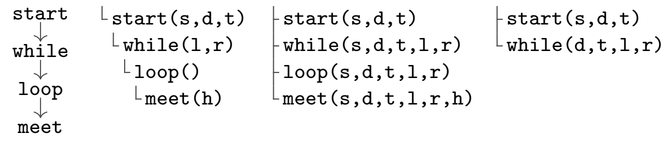
$$\frac{s_1 \equiv s +_\kappa [v \leftarrow q\,;]}{\Gamma \vdash (v := q, s) \mapsto_\kappa (\kappa, s_1)}(\textsc{Assign}) \qquad \frac{\begin{array}{c}\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end} \equiv \textit{new block labels}\\ \Gamma, \langle \kappa_{head}, \kappa_{end} \rangle \vdash (stmts, s) \fatsemi_{\kappa_{body}} (\kappa_1, s_1) \qquad s_2 \equiv s_1 +_\kappa [\texttt{GOTO}\ \kappa_{init}\,;]\end{array}}{\Gamma \vdash (stmts, s) \looparrowright_\kappa ((\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end}), \kappa_1, s_2)}(\textsc{Iter})$$

$$\frac{\begin{array}{c}\Gamma \vdash (stmts, s) \looparrowright_\kappa ((\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end}), \kappa_1, s_1)\\ s_2 \equiv s_1 +_{\kappa_{init}} [\texttt{GOTO}\ \kappa_{head}\,;] +_{\kappa_{head}} [\texttt{GOTO}\ \kappa_{body}\,;] +_{\kappa_1} [\texttt{GOTO}\ \kappa_{head}\,;]\end{array}}{\Gamma \vdash (\texttt{LOOP}\ stmts;\ \texttt{END LOOP}, s) \mapsto_\kappa (\kappa_{end}, s_2)}(\textsc{Loop}) \qquad \frac{\begin{array}{c}\Gamma \vdash (stmts, s) \looparrowright_\kappa ((\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end}), \kappa_1, s_1) \qquad p \equiv \textit{new var}\\ b \equiv [p \leftarrow q\,;,\ \texttt{IF}\ p\ \texttt{THEN GOTO}\ \kappa_{body}\ \texttt{ELSE GOTO}\ \kappa_{end}\,;]\\ s_2 \equiv s_1 +_{\kappa_{init}} [\texttt{GOTO}\ \kappa_{head}\,;] +_{\kappa_{head}} b +_{\kappa_1} [\texttt{GOTO}\ \kappa_{head}\,;]\end{array}}{\Gamma \vdash (\texttt{WHILE}\ q\ \texttt{LOOP}\ stmts;\ \texttt{END LOOP}, s) \mapsto_\kappa (\kappa_{end}, s_2)}(\textsc{While})$$

$$\frac{\begin{array}{c}\Gamma \vdash (stmts, s) \looparrowright_\kappa ((\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end}), \kappa_1, s_1) \qquad p, v_1 \equiv \textit{new vars}\\ b_0 \equiv [v \leftarrow q_0\,;,\ \texttt{GOTO}\ \kappa_{head}\,;]\\ b_1 \equiv [v_1 \leftarrow q_1\,;,\ p \leftarrow v \mathrel{\texttt{<=}} v_1\,;,\ \texttt{IF}\ p\ \texttt{THEN GOTO}\ \kappa_{body}\ \texttt{ELSE GOTO}\ \kappa_{end}\,;]\\ s_2 \equiv s_1 +_{\kappa_{init}} b_0 +_{\kappa_{head}} b_1 +_{\kappa_1} [v \leftarrow v + 1\,;,\ \texttt{GOTO}\ \kappa_{head}\,;]\end{array}}{\Gamma \vdash (\texttt{FOR}\ v\ \texttt{IN}\ q_0\,..\,q_1\ \texttt{LOOP}\ stmts;\ \texttt{END LOOP}, s) \mapsto_\kappa (\kappa_{end}, s_2)}(\textsc{for}) \qquad \frac{s_1 \equiv s +_\kappa [\texttt{GOTO}\ \kappa_{end}\,;]}{\Gamma, \langle \kappa_{head}, \kappa_{end} \rangle \vdash (\texttt{EXIT}, s) \mapsto_\kappa (\kappa, s_1)}(\textsc{Exit}) \qquad \frac{s_1 \equiv s +_\kappa [\texttt{GOTO}\ \kappa_{head}\,;]}{\Gamma, \langle \kappa_{head}, \kappa_{end} \rangle \vdash (\texttt{CONTINUE}, s) \mapsto_\kappa (\kappa, s_1)}(\textsc{Cont})$$

$$\frac{\begin{array}{c}\kappa_{then}, \kappa_{else}, \kappa_{meet} \equiv \textit{new block labels} \quad p \equiv \textit{new var}\\ \Gamma \vdash (stmts_1, s) \fatsemi_{\kappa_{then}} (\kappa_1, s_1) \quad \Gamma \vdash (stmts_2, s_1) \fatsemi_{\kappa_{else}} (\kappa_2, s_2)\\ b \equiv [p \leftarrow q\,;,\ \texttt{IF}\ p\ \texttt{THEN GOTO}\ \kappa_{then}\ \texttt{ELSE GOTO}\ \kappa_{else}\,;]\\ s_3 \equiv s_2 +_\kappa b +_{\kappa_1} [\texttt{GOTO}\ \kappa_{meet}\,;] +_{\kappa_2} [\texttt{GOTO}\ \kappa_{meet}\,;]\end{array}}{\Gamma \vdash (\texttt{IF}\ q\ \texttt{THEN}\ stmts_1\,;\ \texttt{ELSE}\ stmts_2\,;\ \texttt{END IF}, s) \mapsto_\kappa (\kappa_{meet}, s_3)}(\textsc{IfElse}) \qquad \frac{\Gamma \vdash (\texttt{IF}\ q\ \texttt{THEN}\ stmts;\ \texttt{ELSE}\ \varepsilon\ \texttt{END IF}, s) \mapsto_\kappa (\kappa_1, s_1)}{\Gamma \vdash (\texttt{IF}\ q\ \texttt{THEN}\ stmts;\ \texttt{END IF}, s) \mapsto_\kappa (\kappa_1, s_1)}(\textsc{If}) \qquad \frac{s_1 \equiv s +_\kappa [\texttt{RETURN}\ q\,;]}{\Gamma \vdash (\texttt{RETURN}\ q, s) \mapsto_\kappa (\kappa, s_1)}(\textsc{Return})$$

Figure 16: Mapping $\fatsemi$ (and auxiliaries $\mapsto$, $\looparrowright$): Transforming PL/SQL surface syntax into the GOTO-based imperative form.



(a) SSA blocks. **(b) After ANF conversion.** **(c) Functions lifted to top-level.** **(d) Functions inlined.**

Figure 17: Conversion of the block-based SSA form to an ANF function family. (Names `s`, `d`, `t`, `l`, and `h` abbreviate the function arguments `source`, `dest`, `ttl`, `loc`, `route`, and `hop`.)

We determine the arguments of the ANF functions as follows:

- The top-level function `start` receives the arguments of the original UDF (`start(source,dest,ttl)` in Figure 17b).
- SSA discerns data flowing into a block from multiple predecessors in terms of *phi functions*. In ANF, the predecessors (*i.e.*, callers) explicitly pass their data through function arguments instead. ANF conversion thus turns all ϕ-bound variables in block κ into arguments of $\kappa()$ (see variables `loc`, `route` in Lines 4 and 5 of block `while` in Figure 7 and `while(loc,route)` in Figure 17b).

- Scoping rules are the usual: the arguments of a function may be used by its nested functions. Other values passed between blocks—like variable `hop` in Figure 7—are, again, passed as arguments. The body of `meet()` can thus access the arguments `source`, `dest`, `ttl`, `loc`, `route`, and `hop`.

To obtain a family of individual functions whose bodies can be compiled into separate `SELECT` blocks (recall Section 2.4 and see Section 3.2 below), we perform *lambda lifting* [34]. Function arguments that were formerly accessible due to scope nesting, are now passed explicitly. For UDF `route`, this leaves us with the four top-level functions of Figure 17c.

The move to ANF has immediate benefits for the PL/SQL compiler:

- Function inlining now is straightforward. This reduces the number of required function calls and helps to limit trampoline transitions. Inlining `meet()` and `loop()` into `while()` leads to the two-function family of Figure 17d. Further, in `while()`, arguments `source` and `hop` could be removed because they were unused and local, respectively.
- Most importantly, the union of the function argument lists directly determines the arguments of the trampoline and thus the schema of the recursive `UNION` table. (In SSA, this vital information is only implicitly available in terms of the blocks' live variables.) For UDF `route`, this argument union {`source`, `dest`, `ttl`, `loc`, `route`} reduces to {`loc`} since `source`, `dest`, `ttl` are provided by the UDF's calling context and `route` constitutes the UDF's result (already held in column `res`, see Line 1 in Figure 15).

3.2 From ANF to Plain SQL ($\Mapsto$)

Once ANF conversion has left us with the family of tail-recursive functions f_i`()`, we use mapping $\Mapsto$ to compile their bodies e_i into plain SQL (recall Section 2.2). We use $e_i \Mapsto (q_i, t_i)$ to translate ANF expression e_i into a list of SQL range tables (or: "`FROM` list") q_i. In this list, the table ranged over by row variable t_i holds the expression's value so that we can evaluate `SELECT` t_i`.* FROM` q_i to obtain e_i. All tables in the range table list will hold a *single row* (again, this may change if we admit table-valued PL/SQL UDFs, see Section 4.1).

ANF expression e may only take on one of four forms (see Figure 9). Following that, Figure 18 defines $\Mapsto$ by syntactic cases:

- Rule Embed handles embedded SQL subexpression a (this includes variable references, entire scalar SQL subqueries, or the use of built-in function and operators). The rule leaves a intact, wrapping it in a simple `SELECT`.
- Likewise, Rule Call treats tail calls to function $v(a_1, \ldots, a_n)$ (*i.e.*, function name v denotes one f_i of the ANF function family).

Together, the two rules handle the non-recursive (Embed) and tail-recursive (Call) leaf cases of ANF expression evaluation. Since theses cases are treated differently once we perform the conversion to trampolined style, both rules wrap the leaf expressions in $\llbracket\cdot\rrbracket$. We define $\llbracket\cdot\rrbracket$ in Section 3.3 below to ensure that the generated SQL fragments properly plug into the SQL trampoline template.

- Rule Let translates a binding LET $v = a$ IN e. In SQL, we express the evaluation order inherent to this LET: SQL subexpression a is computed first and its value bound to v, then e is evaluated. A `LATERAL` (or dependent) join imposes just this evaluation order [21, 55]. Note that variable scoping in (nested) LETs and SQL's row variable visibility—what is bound left of `LATERAL` is visible to its right—coincide. See below for the role of `LATERAL` in LET compilation. This saves $\Mapsto$ from maintaining its own variable enviroment during the translation.
- Rule Cond compiles an IF-ELSE conditional in terms of SQL's (non-recursive) `UNION ALL`. Since the SQL trampoline uses the same technique to select the correct f_i`()` to dispatch to, let us postpone a discussion until Section 3.3.

To illustrate $\Mapsto$, consider the simple ANF expression e (which computes the area of an n-gon with unit radius, $n \geqslant 3$):

$e \equiv$ LET `tau = 2*3.1415` IN LET `a = tau/`n IN n`/2*sin(a)` .

Compilation via $e \Mapsto (q, \texttt{t2})$ invokes Rules Let (twice) and Embed. If we assume $\llbracket x \rrbracket \equiv x$ for now, this provides us with a range table list q that evaluates e once we wrap it in the SQL `SELECT` block on the right. Note that `t0`, `t1`, `t2`, and the resulting q are singleton tables: the `LATERAL` joins merely communicate variables bindings between expressions and do not inflate the row count. The same is true for `let_p0`, `let_p1`, `let_hop0`, `if_p0`, `if_p1` in Figure 15 and, generally, for any compiled *scalar* PL/SQL UDF. The runtime impact of these singleton joins is negligble.

```
SELECT   t2.*
FROM     (SELECT 2*3.1415)    AS t0(tau),
LATERAL  (SELECT tau/n)       AS t1(a),
LATERAL  (SELECT n/2*sin(a))  AS t2;       q
```

While its internals are organized differently, $\Mapsto$ embodies the main ideas behind *Froid*'s translation [49,53] of non-looping, yet branching PL/SQL UDFs: (1) values bound to a variable v are held in table rows with column v, and (2) dependent joins (here: `LATERAL`, *Froid:* `CROSS APPLY` [21]) are used to express evaluation order. *Froid* collapses multiple independent LETs into a single dependent join and the same optimization applies in our compiler. The apparent proximity of both efforts has already prompted collaboration with the team behind *Froid*.

3.3 Instantiating the SQL Trampoline ($\mathrel{\Vert\!\Rightarrow}$)

Mapping $\mathrel{\Vert\!\Rightarrow}$ calls on $\Mapsto$ and then instantiates the proper SQL trampoline template. This contributes the final missing puzzle piece. $\mathrel{\Vert\!\Rightarrow}$ builds on the family of ANF functions `start` and f_i`()`. Its defining rules of Figure 19 distinguish two scenarios.

Should ANF conversion yield a *one-function family* consisting of function `start(`*params*`)` `:` $\tau = e$ only, the input CFG had no loops and the original PL/SQL UDF `f` comprised a single basic block (recall Section 2.2). This simple case is handled by Rule NonRec of Figure 19 and, in essence, covers *Froid*'s approach [47, 49, 53]. In the absence of looping control flow, no trampolining is required. The rule thus compiles `start`'s body e via $\Mapsto$ and otherwise leaves the obtained `SELECT` block b unaltered (note that the rule defines $\llbracket x \rrbracket \equiv x$).

In the specific formulation of Figure 19, we embed query b in a SQL (*not* PL/SQL) UDF `start()` which can serve as a drop-in replacement for the original PL/SQL UDF `f`. Alternatively, as discussed at the end of Section 2.4, we can replace calls to `f` with the `SELECT` block b directly. Both options are feasible; the choice between the two may depend on the function inlining capabilities of

$$\frac{t \equiv \textit{fresh row var} \qquad q \equiv (\texttt{SELECT}\, [\![a]\!])\ \texttt{AS}\ t}{a \Mapsto (q,\, t)}\,(\textsc{Embed}) \qquad \frac{t \equiv \textit{fresh row var} \qquad q \equiv (\texttt{SELECT}\, [\![v\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{)}]\!])\ \texttt{AS}\ t}{v\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{)} \Mapsto (q,\, t)}\,(\textsc{Call})$$

$$\frac{t_1 \equiv \textit{fresh row var} \qquad q_1 \equiv (\texttt{SELECT}\ a)\ \texttt{AS}\ t_1\texttt{(}v\texttt{)} \qquad e \Mapsto (q_2,\, t_2)}{\texttt{LET}\ v = a\ \texttt{IN}\ e \Mapsto (q_1\, \texttt{,LATERAL}\ q_2\ ,\ t_2)}\,(\textsc{Let})$$

$$\frac{t \equiv \textit{fresh row var} \qquad e_1 \Mapsto (q_1,\, t_1) \qquad e_2 \Mapsto (q_2,\, t_2) \qquad b_1 \equiv \texttt{SELECT}\ t_1\texttt{.*}\ \texttt{FROM}\ q_1\ \texttt{WHERE}\ v \qquad b_2 \equiv \texttt{SELECT}\ t_2\texttt{.*}\ \texttt{FROM}\ q_2\ \texttt{WHERE NOT}\ v}{\texttt{IF}\ v\ \texttt{THEN}\ e_1\ \texttt{ELSE}\ e_2 \Mapsto (\texttt{(}b_1\ \texttt{UNION ALL}\ b_2\texttt{)}\ \texttt{AS}\ t,\ t)}\,(\textsc{Cond})$$

Figure 18: Mapping $\Mapsto$: Translating ANF expressions into SQL range tables (or "FROM list"). Auxiliary $[\![\cdot]\!]$ is defined in Figure 19.

the host RDBMS and the amount of query code duplication that we are willing to tolerate.

Rule Rec of Figure 19 covers the general, looping case in which we are handed a family of functions start(), f_1(), …, f_n(). This rule instantiates the SQL trampoline first introduced in Figure 14. The core of the trampoline is a recursive CTE whose working table run holds the single row resulting from the evaluation of the last f_i() (or, initially, start()) function: while column res contains the function's actual result, columns "rec?" and call determine the trampoline's next action (Section 2.3). Template instantiation proceeds as follows:

(1) Translate the bodies $e_0, \ldots, e_n$ of start and the f_i() via $\Mapsto$, obtain range lists $q_0, \ldots, q_n$.
(2) Wrap the $q_1, \ldots, q_n$ in SELECT blocks $b_1, \ldots, b_n$. Evaluation of b_i is conditioned on predicate run.call = 'f_i'. Block b_0 for q_0 (function start()) is evaluated unconditionally.
(3) For all b_i, define $[\![\cdot]\!]$ so that the trampoline acts on their result:
   - to return value a, set column "rec?" to false and res to a,
   - to tail-call $v(a_1, \ldots, a_m)$, set "rec?" to true, call to 'v', and set the function argument column v_j to a_j ($j = 1, \ldots, m$).
(4) Use b_0 as the initializing query $\texttt{Q}_{\texttt{start}}$ (cf. Figure 11c) of the recursive CTE. Place $b_1, \ldots, b_n$ in the CTE's recursive part $\texttt{Q}_{\texttt{rec}}$, forming an n-fold dispatcher based on a stack of (non-recursive) UNION ALLs.

Placing the resulting recursive CTE *trampoline* in SQL UDF start() (see above) completes function compilation.

Compiling multi-way conditionals. The trampoline's dispatcher and Rule Cond of $\Mapsto$ (Figure 18) rely on a stack of $n-1$ UNION ALLs to implement n-fold case distinction between query blocks $b_1, \ldots, b_n$ (in the case of Rule Cond, $n = 2$). Consider b_1 UNION ALL b_2 in which the b_i contain mutually exclusive WHERE predicates p_i that are *independent* of their block (see Rules Cond and Rec). On PostgreSQL, this translates into the physical plan shown on the left. In such plans, operators RESULT evaluate the so-called *one-time filters* p_i once *before* the sub-plans for the block b_i are processed [45]. Should p_i turn out false, the plan for b_i is never entered. This plan behavior exactly implements the expected "lazy" behavior of an IF-ELSE conditional (or an entire stack of such conditionals if $n > 2$). We have found this to be a compositional and performant translation of conditionals on PostgreSQL, version 11 and beyond. Other RDBMSs exploit similar operator configurations and runtime behavior (*e.g.*, in Oracle 19c [42], a UNION-ALL/FILTER pair ensures that *either* b_1 *or* b_2 is evaluated).

```
APPEND
├RESULT [p1]
│ └⟨plan for b1⟩
└RESULT [p2]
  └⟨plan for b2⟩
```

Local changes to the SQL code emitted by $\Mapsto$ and $\Mmapsto$ should be able to accomodate the specifics of a wide range of RDBMS targets.

Translating PL/SQL UDFs with effects. Throughout, we expected PL/SQL UDF f to be a *pure* function. UDFs with side effects—like database updates, SQLSTATE changes, or exceptions—can be embraced by an *encode-then-perform* approach:

Encode: Stick to the compilation strategy but translate a side-effecting statement into a SELECT block b_i (see Sections 2.4 and 3.2) returning a row that *encodes* the effect. To this end, extend the trio of columns rec?,call,res (Figure 14) by a column eff.

Perform: In a thin PL/SQL wrapper, say $\texttt{f}^+$, first evaluate the emitted SQL code for f. Inspect column eff in f's result and *perform* the encoded effects, *i.e.*, issue row updates or raise an exception.

This two-phase evaluation is reminiscent of XQuery's *pending update lists* [54] or Haskell's monadic I/O subsystem [31]. At runtime, we would see $\texttt{Q} \to \texttt{f}^+$ context switches but still save all $\texttt{f} \to \texttt{Q}$ overhead. As is expected in the context of effects, the need for a PL/SQL interpreter remains, however. Our exploration of the compilation of side-effecting UDFs is ongoing.

4 EXPERIMENTS WITH 18 PL/SQL UDFs

The deliberately simple PL/SQL UDF route with its single embedded SQL query $\texttt{Q}_1$ already uncovered the challenges of repeated $\texttt{route} \leftrightarrows \texttt{Q}_1$ context switching. If we turn up UDF complexity, the situation becomes even more dire. This section studies 18 PL/SQL UDFs of low to extremely high complexity, and quantifies the runtime and memory impact that compilation to plain SQL can have on these UDFs. All of the following measurements have been performed with PostgreSQL 11.3 [45], our primary experimentation platform. The RDBMS was hosted on a 64-bit Linux x86 host (8 Intel Core™ i7 CPUs at 3.66 GHz, 64 GB of RAM). Timings were averaged over five runs, with worst and best times disregarded. We exclude the one-shot UDF compilation times of about 250 ms.

Compiling a collection of UDFs. Table 1 lists the 18 UDFs, for which the original PL/pgSQL source, the compiled SQL counterparts, and sample input data can be found on *GitHub*.[3] The UDFs implement a wide range of algorithms—*e.g.*, routines in 2D/3D geometry, the simulation of VMs and spreadsheets, or optimization problems over TPC-H data—on a variety of built-in and user-defined data types (see column **Return type**). Columns **LOC** (lines of code), $|\texttt{Q}_i|$ (number of non-fast-path embedded SQL queries), and **Loop constructs** aim to characterize the UDFs in terms of code size and loop nesting. Recall that the number of loops determines the size of the ANF function family: ray comprises four loops (nested

[3] http://github.com/One-WITH-RECURSIVE-is-Worth-Many-GOTOs. The files for UDF f reside in subdirectory f/ of the repository.

$$[\![x]\!] \equiv x$$

$$\frac{e \Mapsto (q,t) \qquad b \equiv \texttt{SELECT}\ t.*\ \texttt{FROM}\ q}{[\texttt{start}(\mathit{params}) : \tau = e] \quad \Mapsto \quad \begin{array}{l}\texttt{CREATE FUNCTION start}(\mathit{params})\ \texttt{RETURNS}\ \tau\ \texttt{AS}\\ \texttt{\$\$}\ \ b\ \ \texttt{\$\$ LANGUAGE SQL;}\end{array}}\ (\textsc{NonRec})$$

$$\begin{array}{rcl}[\![a]\!] & \equiv & \texttt{false AS "rec?", NULL AS call, } a \ \ \texttt{AS res, NULL AS } v_1, \ldots, \texttt{NULL AS } v_m\\ [\![v(a_1,\ldots,a_m)]\!] & \equiv & \texttt{true AS "rec?", '}v\texttt{' AS call, NULL AS res, } a_1 \ \ \texttt{AS } v_1, \ldots, a_m \ \ \texttt{AS } v_m\end{array}$$

$$\frac{\begin{array}{l} e_i \Mapsto (q_i, t_i)|_{i=0\ldots n} \qquad b_0 \equiv \texttt{SELECT}\ t_0.*\ \texttt{FROM}\ q_0 \qquad b_i \equiv \texttt{SELECT}\ t_i.*\ \texttt{FROM}\ q_i\ \texttt{WHERE run.call = '}f_i\texttt{'}|_{i=1\ldots n}\\ \mathit{trampoline} \equiv \texttt{WITH RECURSIVE run("rec?",call,res,}v_1,\ldots,v_m\texttt{) AS (}\\ \qquad b_0\\ \qquad\quad \texttt{UNION ALL}\\ \qquad \texttt{SELECT\ \ result.*}\\ \qquad \texttt{FROM\ \ \ \ run,}\\ \qquad \texttt{LATERAL (}b_1\ \texttt{UNION ALL} \cdots \texttt{UNION ALL}\ b_n\texttt{) AS result}\\ \qquad \texttt{WHERE\ \ \ run."rec?"}\\ \qquad \texttt{)}\\ \qquad \texttt{SELECT run.res FROM run WHERE NOT run."rec?"}\end{array}}{\left[\begin{array}{ll}\texttt{start}(\mathit{params}) & : \tau = e_0,\\ f_1(v_1\ \tau_1,\ldots,v_m\ \tau_m) & : \tau = e_1,\\ \qquad\vdots & \\ f_n(v_1\ \tau_1,\ldots,v_m\ \tau_m) & : \tau = e_n\end{array}\right] \quad \Mapsto \quad \begin{array}{l}\texttt{CREATE FUNCTION start}(\mathit{params})\ \texttt{RETURNS}\ \tau\ \texttt{AS}\\ \texttt{\$\$}\ \ \mathit{trampoline}\ \ \texttt{\$\$ LANGUAGE SQL;}\end{array}}\ (\textsc{Rec})$$

Figure 19: Mapping ⤇: Instantiating the SQL-based trampoline. Receives ANF function family [start(), f_i()], emits plain SQL.

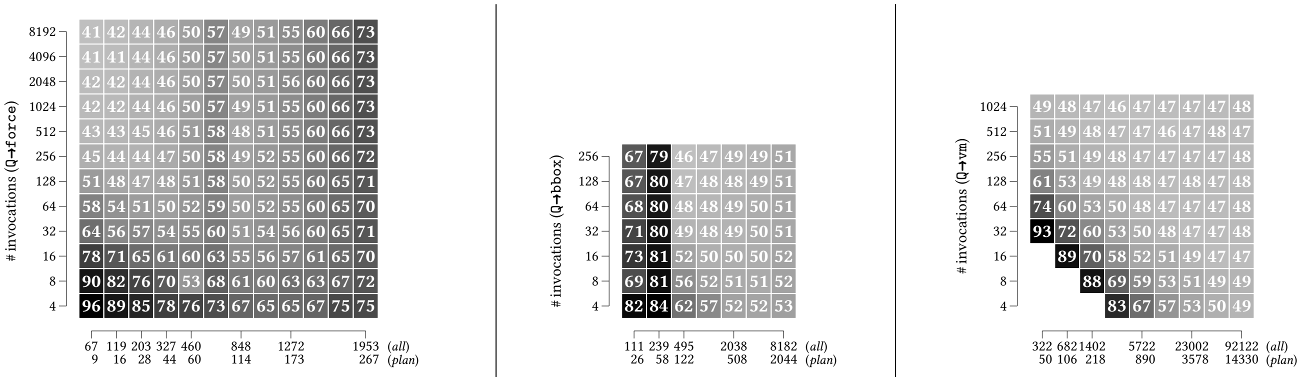
Table 1: A collection of PL/SQL UDFs with context switching overhead before and speedup after compilation to SQL.

UDF		Return type	|Q$_i$|	LOC	Loop constructs	CC	Q→f+f→Q$_i$ overhead	Runtime (speedup) after compilation		Trampoline transitions
bbox	detect bounding box of a 2D object	box	2	41	1 ⎵	5	25.0%	48.0%	(2.08 ×)	1157
force	*n*-body simulation (Barnes-Hut quad tree)	point	3	43	1 ⎵	5	40.7%	49.0%	(2.04 ×)	263
global	does TPC-H order ship intercontinentally?	boolean	1	20	1 ⎵	3	32.8%	59.0%	(1.69 ×)	9
items	count items in hierarchy (adapted from [27])	int	2	27	1 ⎵	2	48.3%	33.0%	(3.03 ×)	4097
late	find delayed orders (transcribed TPC-H *Q*21)	boolean	1	25	1 ⎵	4	28.8%	66.0%	(1.51 ×)	9
march	track border of 2D object (Marching Squares)	point[]	2	40	1 ⎵	5	27.2%	73.0%	(1.36 ×)	4568
margin	buy/sell TPC-H orders to maximize margin	row	3	57	1 ⎵	5	17.4%	87.0%	(1.14 ×)	59
markov	Markov-chain based robot control	int	3	44	1 ⎵	3	13.8%	77.0%	(1.29 ×)	1026
packing	pack TPC-H lineitems tightly into containers	int[][]	3	82	3 ⎵⎵⎵	9	43.1%	56.0%	(1.78 ×)	312
savings	optimize supply chain of a TPC-H order	row	6	66	1 ⎵	4	35.7%	38.0%	(2.63 ×)	9
sched	schedule production of TPC-H lineitems	row array	5	77	2 ⎵⎵	6	29.7%	63.0%	(1.58 ×)	33
service	determine service level (taken from [53])	text	1	22	0 ∅	3	28.2%	55.0%	(1.81 ×)	0
ship	customer's preferred shipping mode	text	3	34	0 ∅	3	14.7%	75.0%	(1.33 ×)	0
sight	compute polygon seen by point light source	polygon	3	49	2 ⎵⎵	3	61.5%	82.0%	(1.21 ×)	662
visible	derive visibility in a 3D hilly landscape	boolean	2	57	1 ⎵	3	26.8%	49.0%	(2.04 ×)	258
vm	execute program on a simple virtual machine	numeric	1	28	1 ⎵	17	37.3%	48.0%	(2.08 ×)	7166
ray	complete PL/SQL ray tracer (adapted from [30])	int[]	5	230	4 ⎵⎵⎵⎵	25	12.9%	422.0%	(0.23 ×)	59642
sheet	evaluate inter-dependent spreadsheet formulæ	float	9	117	4 ⎵⎵⎵⎵	19	22.7%	104.0%	(0.96 ×)	2811

inside each other) which yields functions start(), f_1(), …, f_4() after ANF conversion, for example. The UDFs service and ship contain branching control flow but do not iterate at all. Their ANF conversion yields a single start() function, a case covered by Rule NonRec of Figure 19. No trampoline or recursive CTE is generated. *Froid* [49] would emit similar SQL code for these two UDFs. **CC** (cyclomatic complexity) counts the number of independent control flow paths [38] through the PL/SQL UDFs and constitutes a measure of control flow complexity (LOC and CC for route are 22 and 3, respectively).

UDF compilation from PL/SQL to SQL follows the hypothesis that we can save Q→f and f→Q$_i$ context switching effort if we operate entirely on the SQL side of the fence. Column Q→f+f→Q$_i$ **overhead** quantifies this effort which may even exceed the time invested in actual UDF evaluation (61.5% overhead for UDF sight). Compilation to plain SQL can entirely remove this overhead and may improve runtime beyond that: for example, PostgreSQL invested about a good 1/3 of the evaluation time for PL/SQL UDF savings into context switching, but the function runs 2.63 times faster after compilation, see column **Runtime (speedup) after compilation**. This is the performance advantage we were after with compilation.

The outcome is different for UDFs ray and sheet which exhibit slowdown after compilation. We have included these functions foremost to demonstrate that the techniques of Sections 2 and 3 scale to super-complex control flow—such UDFs would be seldomly found in practice and do stretch the idea of computation close to the data. Performance suffers due to (a) the construction of large intermediate data structures, *e.g.*, arrays of 1800+ pixels in the case of ray, (b) extensive sets of live variables (44 for ray),



(a) Heat map for UDF force.

(b) Heat map for UDF bbox.

(c) Heat map for UDF vm.

Figure 20: Runtime (in % of PL/SQL UDFs) after compilation, Q→f and f→Q_i context switches varied (lower/lighter is better).

and (c) complex control flow that leads to a high number of transitions through the trampoline (Column **Trampoline transitions**). While (a) and (b) affect the width, (c) determines the cardinality of the UNION table built by the recursive CTE. If we manage, however, to keep the resulting memory pressure on the system's buffer cache in check, things look considerably brighter even for these super-complex UDFs (see Section 4.1).

Non-looping and trampoline-free UDFs service and ship aside, each transition through the trampoline that we can avoid constitutes a double win: WITH RECURSIVE needs to perform fewer iterations and consequently appends fewer rows to the resulting UNION table. This underscores the importance of optimizations that reduce the size of the ANF function family: function inlining saves tail calls, each of which transition through trampoline() once. To exemplify, the ANF function family for packing has $1 + 3$ functions (see Figure 11a). With function inlining disabled, the family consists of 6 functions and both, the number of transitions through trampoline() (from 312 to 608) as well as the UNION table size (from 52 to 103 kB), nearly double.

Varying the impact of Q→f and f→Q_i. In the heat maps of Figure 20, we focus on the UDFs force, bbox, vm and vary the number of context switches (Q→f: UDF invocations, f→Q_i: evaluation of embedded SQL queries per invocation) performed during the experimental runs. (Table 1 reports on the runs in the center of these heat maps where the impact of Q→f and f→Q_i is expected to be average.) We adapt a top-level SQL query (cf. Q_0 of Section 1) to vary the number of UDF invocations. Likewise, we alter the function's input size to vary the number of intra-function iterations and thus the number of embedded SQL queries evaluated per invocation. This provides some, yet not exact, control over the total of f→Q_i context switches and explains the irregular f→Q_i axes in the heat maps (for these axes, *all* refers to all such switches, while *plan* counts the subset of those that required query planning and did not qualify for fast-path evaluation). The heat map entries show the runtime needed after compilation to SQL, relative to the original PL/SQL UDF: 46 indicates that the compiled UDF executed in 46% of the original runtime, lower/lighter is better. For these three UDFs, the absolute runtimes per invocation ranged from 0.2 ms (leftmost heat map column) to 30 ms (rightmost column) after compilation.

Following one heat map column bottom to top, we find that compilation saves the more runtime the more UDF invocations are performed: Q→f overhead is avoided repeatedly. The effect is particularly pronounced if little work is performed inside f (*i.e*, to the left of the maps) since PL/SQL sees less opportunity to compensate the Q→f context switch costs.

The compiled UDF force (Figure 20a) requires as low as 41% of the original runtime if lots of invocations are performed. As the input size and thus the number intra-function iterations grow, this goes up to 73% of the PL/SQL runtime. For force, input size is determined by the number of bodies represented by an underlying Barnes-Hut [4] quad tree. More bodies lead to deeper tree descents and to larger intermediate data structures (force performs breadth-first traversal and saves nodes left to explore in a SQL array.) Both contribute to a growing working table size (also see Table 2 below) which impacts the compiled function's runtime.

The notable dark-light boundary at 239 bbox→Q_i embedded SQL evaluations in the heat map for UDF bbox (Figure 20b) is due to an optimizer plan change: PostgreSQL switches from a SEQ SCAN to an INDEX SCAN with growing input table size which leads to a significant runtime reduction for the compiled function. Generally, PL/SQL UDFs cannot benefit from such optimizer decisions: so-called *generic plans* for the Q_i are determined and fixed when the embedded SQL queries are first encountered [45, §42.11]. The picture is different after compilation. Two embedded SQL queries in bbox—originally separate in the PL/SQL UDF, with the result of the first query bound to a variable and passed on to the second—are planned and optimized as a whole (leading to one join) in the UDF's SQL variant. Further, the optimizer is aware of input table sizes and can adapt its plan accordingly. Such beneficial multiple-query optimization effects [52] explain runtime improvements beyond the saved context switch overhead (recall Table 1).

The original PL/SQL variant of UDF vm performs a vm→Q_1 context switch for each instruction executed by the simulated VM (*e.g.*, the top-right run in the heat map for vm performs about 1024 ×

Table 2: WITH RECURSIVE *vs.* WITH ITERATE: Memory allocation and buffer page writes due to UNION table construction.

UDF	WITH RECURSIVE			WITH ITERATE		
	UNION table	# writes 512 kB	# writes 100 MB	working table	# writes	Speedup over RECURSIVE
bbox	122 kB	55 793	0	161 b	0	marginal
force	1 155 kB	633 756	0	81 b	0	
global	789 b	0	0	85 b	0	
items	448 kB	100	0	69 b	0	
late	10 kB	0	0	1192 b	0	
margin	7 kB	0	0	121 b	0	
markov	40 kB	0	0	41 b	0	
packing	52 kB	0	0	580 b	0	
savings	2 kB	0	0	714 b	0	
sched	21 kB	0	0	1088 b	0	
sight	3 544 kB	14 143	0	15 kB	0	
visible	40 kB	0	0	152 b	0	
vm	1 643 kB	198 849	0	432 b	0	
march	256 MB	33.5 M	33.5 M	256 kB	0	2.96 ×
ray	244 MB	2 M	2 M	15 kB	0	1.37 ×
sheet	100 MB	6.5 M	6.5 M	62 kB	0	3.17 ×

14330 vm→Q_1 switches, see Figure 20c). Q_1 itself is simple and executes quickly, such that the massive vm→Q_1 context switch costs can never be compensated. This is a best-case scenario for the compilation to SQL where no context switch is incurred at all. The UNION table built by vm's trampoline is sizable (we turn to these space aspects in Section 4.1). Still, the resulting savings dominate the effort to evaluate the recursive CTE. A compiled vm indeed executes at about twice the speed—≈ 48% of the original runtime—in the top-right half of the heat map. The closer we approach the lower-left corner of the map and the fewer context switch saving opportunities present themselves, the less differ the runtimes of the original and compiled functions. We have omitted the measurements in the very lower-left where the runtimes of both were in the sub-millisecond range and effectively on par.

4.1 To Recurse is Divine, to ITERATE Space-Saving

The evaluation of the recursive CTE that forms the SQL trampoline builds a UNION table run containing a trace of all ANF function calls (Section 2.4). While this stack-like trace can be insightful and aid function debugging [29], its construction may consume significant buffer space. This is particularly so for compiled functions that maintain many live variables of sizable contents (leading to *wide* UNION tables) and transition through the trampoline often (*long* UNION tables, containing one row per transition). We have recorded the resulting UNION table size in Table 2 under heading WITH RECURSIVE. We have omitted the non-looping UDFs service and ship since they allocate no UNION table at all. Table 2 reports on worst-case runs in which the functions processed large inputs, corresponding to the right-most columns in the heat maps of Figure 20. Should the UNION table exceed the buffer space allocated for the evaluation of WITH RECURSIVE, PostgreSQL maintains the entire table in secondary memory. The number of incurred page write operations, for buffer sizes of 512 kB and 100 MB, are also found in Table 2. UDFs with high totals of trampoline transitions (ray, vm) and/or significant live variable sets (march, ray, sheet; the latter manipulates JSON representations of spreadsheet formulæ, for example) stick out in particular. This may be countered by noticing that all rows but the one with "rec?" = false are discarded from the UNION table before the final function result is returned (cf. predicate NOT "rec?" in Line 14 of Figure 14): the stack-like trace of the computation is never consumed and need not be built in the first place. (In the parlance of programming languages, since we perform tail calls only, there is no need for a stack.)

We built this behavior directly into PostgreSQL 11.3 in terms of WITH ITERATE, a variant of WITH RECURSIVE that holds on to the last row of the working table only [14, 29, 44]. No UNION table is built. Effectively, the predicate NOT "rec?" is pushed down into the computation of the recursive CTE—otherwise its semantics remain as is. In a Volcano-style query execution engine [24], WITH ITERATE does not return to its parent operator before the one result row is produced. While this only marginally reduces query evaluation time, the space savings can be significant. We only ever allocate space for a single working table row and, regardless of buffer size, no page I/O is required (see the columns under heading WITH ITERATE in Table 2). Exactly those compiled UDFs that were penalized the most by UNION table construction do benefit. A one-line change from WITH RECURSIVE to WITH ITERATE in the SQL trampoline template of Figure 14 reinstates the low memory requirements of the original PL/SQL UDF: the single working table row holds exactly one copy of the contents of all live variables. That row—occupying mere bytes or few kilobytes—easily fits into main memory and the complex UDFs march and sheet are sped up by a factor of 3 (ray benefits less due to high number of transitions through trampoline() that remain).

The runtime and space savings due to WITH ITERATE can be substantial. Yet its implementation reaches inside the PostgreSQL kernel, defying the ideal of a pure source-to-source compilation approach (Section 1.1). We thus consider WITH ITERATE an optimization rather than part of the core design. The results, however, have prompted use to consider other local database kernel changes that aid PL/SQL compilation (*e.g.*, cursor access from within SQL queries).

Table-valued UDFs. One can argue that the construction of sizable table-like data structures in PL/SQL array variables runs counter the relational school (and, indeed, the system penalizes this style as just discussed above). In some scenarios, *table-valued UDFs* offer an alternative style: instead of appending elements to ever-growing arrays, we use PL/SQL's RETURN NEXT a to emit the function's result row-by-row [45, §43.6]. Note that this statement does *not* alter the UDF's control flow: execution continues immediately after RETURN NEXT so that the UDF may emit further result rows.

Integrating these semantics into the UDF compiler calls for a local change to the $\Rightarrow$ mapping of Figure 18. In a nutshell, a new rule will generate a SQL SELECT block b_i that returns *multiple* rows. All of these rows will be tagged with "rec?" = false and thus find their way into the UNION table and thus the function's final result. This behavior readily fits with the WITH RECURSIVE-based trampoline of Figure 14. The move to a table-valued UDF alleviates the need for sizable array data held in PL/SQL variables: working table width shrinks (for UDF march, from an average of 256 kB down to 81 b). Additionally, the UDF saves repeated array copy and append (`||`) operations. A table-valued compiled UDF `march`, for example, sees tangible performance benefits and runs in 57% of the runtime of the original table-valued PL/SQL UDF (down from 73% for a scalar UDF `march`, see Table 1). We include both, scalar and table-valued, variants of `march` in the *GitHub* repository. A complete study of table-valued UDFs is currently in progress.

5 MORE RELATED WORK

With PL/SQL, there is no dividing line between where the data lives and where computation takes place: the PL/SQL interpreter sits right inside the database kernel, with immediate access to the base and meta data. It is the tension between imperative/iterative and the declarative/set-oriented modes of execution that spoil the promising PL/SQL idea [14, 49]. The need for complex computation close to the data [50] will only ever go up, as exemplified by contemporary efforts to host machine learning algorithms inside RDBMSs [5, 17, 33]. Such computation can be expressed using regular imperative PLs and then be compiled into queries [16, 25], but we believe that PL/SQL with its wide availability, SQL-native type system, and seamless embedding of SQL queries is particularly worth to study (and to improve!).

In this respect, we side with recent efforts like *Froid* [47–49] that try to find new ways to blend the execution of imperative T-SQL code with plan-based SQL evaluation. The original *Froid* effort was restricted to branching, but non-looping control flow which puts 16 out of 18 UDFs in Table 1 out of *Froid*'s reach. We consider support for iteration (or recursion schemes, possibly even nonlinear [13]) essential for any effort to host complex computation inside a RDBMS.

The impact of `Q→f` and `f→Q` context switches that result from the iterated invocation of UDFs and embedded queries can be reduced through *batching*. For a language close to our PL/SQL dialect of Figure 5—yet without `CONTINUE` or `EXIT`—Guravannavar *et.al.* [27] describe how to lift iterated operations over scalars to non-iterated table-valued computation. This batching transformation is versatile but it (1) constructs nested intermediate tables from nested loops, thus depending on table *(un)nest* operations, and (2) fails to batch cyclic loop dependencies. UDF `route` of Figure 2 contains such a dependency between `hop`/`loc` in Line 11/Line 17, and thus would not be subject to batching, for example. Batched UDFs still are PL/SQL UDFs, such that a (reduced) number of context switches as well as the general need for a PL/SQL interpreter remain.

With *Aggify* [26], the *Froid* team pursues a follow-up approach that can embrace limited forms of looping, provided that the number of iterations is known in advance (a conditional early `EXIT` thus could not be used to stop iteration, for example). *Aggify* translates a loop into a user-defined SQL aggregate and places the loop body inside the aggregate's accumulator routine, a technique first proposed in [53]. Details on the treatment of nested loops are not spelled out in [26,53]. We believe that the trampoline-based PL/SQL compilation with its inherent support for *entirely arbitrary* control flow really has an edge here.

In a sense, iterated accumulation assumes the role of repeated trampoline evaluation performed by `WITH RECURSIVE`. Since both approaches appear to be compatible otherwise, on a trial basis we embedded our dispatcher (essentially the lines Lines 6 to 11 of Figure 14) inside a user-defined PostgreSQL aggregate [45, §38.11] instead. Runs for UDF `savings`—a UDF whose control flow proved sufficiently simple for the approach—took about 5 × longer once we replaced the original SQL trampoline with the aggregate: while the top-level query evaluates the aggregate, PostgreSQL 11.3 needs to repeatedly switch to the plan comprising the accumulator body. The resulting back and forth between the top-level and accumulator plans—both remain separate—slows the system down (we have discussed just this phenomenon in Section 1). Let us note that the above does *not* constitute a faithful re-implementation of *Aggify* on PostgreSQL: in [26] it is proposed to implement the accumulator body in *C#* (which we regard out of scope for our line of work that targets plain SQL). A pure-SQL alternative may be *flattening* [57], a query transformation that can handle loops of the above restricted kind and tightly integrates the top-level and loop body queries.

We have developed a sketch and an initial implementation of a PL/SQL to SQL compiler that, too, pursues the reduction of costly `Q→f` and `f→Q` transitions in [14,29]. With that approach, the present work shares the first compilation stages up to the generation of an ANF function family. In [14], we then deviate and propose to (1) *directly* translate the f_i`()` into mutually tail-recursive plain SQL UDFs, before we (2) identify the tail calls in these UDFs to derive a recursive CTE that inlines and dispatches to the UDFs' bodies. While step (1) provides an immediately executable intermediate form on RDBMSs that support such recursive UDFs (like PostgreSQL), we have found its function call overhead at runtime to be substantial. The trampoline-based approach avoids this intermediate form and provides a principled alternative to the *ad hoc* step (2)—neither [14] nor [29] recognized the connection to trampolined style. With that link now established, trampoline-based generalizations (*e.g.*, UDF evaluation with scheduled interruption and resumption) and optimizations (*e.g.*, the exploitation of parallelism) are within reach [22].

In [14], the translation itself and the associated experiment were exercised using a single PL/SQL UDF. The present work is the first to lay out all details of the compilation pipeline (Sections 2.3, 2.4, and 3) as well as to study a broad class of UDFs that exhibit widely varying control flow complexities and `UNION` table sizes (regarding table width as well as length, Section 4).

The PL/SQL compiler does not depend on a specific SSA form. Our simple, homegrown variant could thus be replaced by an implementation with full-fledged optimization passes, as provided by *MLIR* [36], for example. *MLIR*, in fact, employs *parameterized* SSA blocks which make inter-block data flow explicit (a prerequisite to derive the `UNION` table schema, recall Section 3.1). This renders *MLIR*'s particular intermediate representation close to our ANF. Since the primary translation target for *MLIR* is LLVM code, a new SQL-emitting backend would be required to make this alternative implementation avenue viable, however.

We assume that PL/SQL UDFs are being *interpreted*, as implemented by PostgreSQL [45], Oracle [42], or SQL Server [39], for example. A new breed of RDBMSs compiles UDFs to byte code [2,35] or native machine instructions [10,41]. These systems rely on SQL queries being compiled in the same fashion (otherwise, the compiled code would need to invoke the regular plan engine to evaluate the embedded SQL queries $\mathtt{Q}_i$, reintroducing context switching friction). If translation from PL/SQL to SQL is performed *before* SQL code generation, the present work could find a place in these recent systems and render their compilers considerably simpler. We are keen to explore this next.

6 WRAP-UP

Programming in PL/SQL is imperative programming. This steered us towards techniques devised by the programming language community to implement a compiler that emits plain SQL. SSA, ANF conversion, and trampolined style are long-established and provide a solid foundation that does not crack under the weight of very complex PL/SQL UDFs. Translating trampolined style in terms of SQL's `WITH RECURSIVE` (or `ITERATE`) is our attempt to bring the power of plan-based query evaluation to bear on iterative data-intensive computation. We are positive that the compilation technique is practical and widely applicable as it is non-invasive in a double sense: (1) any RDBMS with support for contemporary SQL (recursive CTEs and `LATERAL`) is a viable target platform and (2) compiled as well as regular PL/SQL UDFs may coexist in one database application and call each other (UDFs may be compiled selectively if they turn out to be performance bottlenecks).

As we write this, we extend this study to the translation of table-valued UDFs and the integration into RDBMSs that pursue SQL (but not PL/SQL) compilation. On both fronts, preliminary results are certainly promising.

REFERENCES

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

[2] M. Armbrust, R.S. Xin, C. Lian, Y. Huai, D. Liu, J.K. Bradley, X. Meng, T. Kaftan, M.J. Franklin, and A. Ghodsi. Spark SQL: Relational Data Processing in Spark. In *Proc. SIGMOD*, Melbourne, Australia, May 2015.

[3] F. Bancilhon. Naive Evaluation of Recursively Defined Relations. In *On Knowledge Base Management Systems*, pages 165–178. Springer, 1986.

[4] J. Barnes and P. Hut. A Hierarchical $O(N \log N)$ Force-Calculuation Algorithm. *Nature*, 324(4), 1986.

[5] M. Boehm, A. Kumar, and J. Yang. *Data Management in Machine Learning Systems*. Synthesis Lectures on Data Management. Morgan & Claypool, 2019.

[6] M. Braun, S. Buchwald, S. Hack, R. Leiša, C. Mallon, and A. Zwinkau. Simple and Efficient Construction of Static Single Assignment Form. In *Proc. Int'l Conference on Compiler Construction*, Rome, Italy, March 2013.

[7] TIOBE Software BV. TIOBE Index for the PL/SQL Programming Language, 2020. https://www.tiobe.com/tiobe-index/pl-sql/.

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

[9] P. P. Chang and W.-W. Hwu. Inline Function Expansion for Compiling C Programs. In *Proc. PLDI*, Portland, OR, USA, June 1989.

[10] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvili, and M. Andrews. The MemSQL Query Optimizer: A Modern Optimizer for Real-Time Analytics in a Distributed Database. *Proc. VLDB*, 9(13), 2016.

[11] D. Cooper. Böhm and Jacopini's Reduction of Flow Charts. *Communications of the ACM*, 10(8), August 1967. Letter to the Editor.

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

[13] C. Duta and T. Grust. Functional-Style UDFs With a Capital 'F'. In *Proc. SIGMOD*, Portland, OR, USA, June 2020.

[14] C. Duta, D. Hirn, and T. Grust. Compiling PL/SQL Away. In *Proc. CIDR*, Amsterdam, The Netherlands, January 2020.

[15] A. Eisenberg and J. Melton. SQL:1999, Formerly Known as SQL3. *ACM SIGMOD Record*, 28(1), March 1999.

[16] K.V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proc. SIGMOD*, San Francisco, CA, USA, June 2016.

[17] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *Proc. SIGMOD*, Scottsdale, AZ, USA, May 2012.

[18] J. Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM TOPLAS*, 9(3), July 1987.

[19] S.J. Finkelstein, N. Mattos, I. Mumick, and H. Pirahesh. Expressive Recursive Queries in SQL. Joint Technical Committee ISO/IEC JTC 1/SC 21 WG 3, Document X3H2-96-075r1, March 1996. https://kirusa.com/mumick/pspapers/ansiRevisedRecProp96-075r1.ps.Z.

[20] C. Flanagan, A. Sabry, B.F. Duba, and M. Felleisen. The Essence of Compiling with Continuations. *ACM SIGPLAN Notices*, 28(6), June 1993.

[21] C. Galindo-Legaria and M. Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *Proc. SIGMOD*, 2001.

[22] S.E. Ganz, D.P. Friedman, and M. Wand. Trampolined Style. In *Proc. ICFP*, Paris, France, September 1999.

[23] J.J. Garcia-Luna-Aceves and S. Murthy. A Path-Finding Algorithm for Loop-Free Routing. *IEEE/ACM Transactions on Networking*, 5(1), February 1997.

[24] G. Graefe. Volcano—An Extensible and Parallel Query Evaluation System. *IEEE TKDE*, 6(1), February 1994.

[25] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-Safe LINQ Compilation. *Proc. VLDB*, 3(1), September 2010.

[26] S. Gupta, S. Purandare, and K. Ramachandra. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *Proc. SIGMOD*, Portland, OR, USA, June 2020.

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

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

[29] D. Hirn and T. Grust. PL/SQL Without the PL. In *Proc. SIGMOD*, Portland, OR, USA, June 2020.

[30] Holtsetio. MySQL Raytracer, October 2019. https://demozoo.org/productions/268459/.

[31] P. Hudak, J. Hughes, S. Peyton-Jones, and P. Wadler. A History of Haskell: Being Lazy with Class. In *Proc. HOPL-III*, San Diego, CA, USA, June 2007.

[32] S. Jagannathan and A. Wright. Flow-Directed Inlining. *ACM SIGPLAN Notices*, 31(5), May 1996.

[33] D. Jankov, S. Luo, B. Yuan, Z. Cai, J. Zou, C. Jermaine, and Z.J. Gao. Declarative Recursive Computation on an RDBMS (or Why You Should Use a Database for Distributed Machine Learning). *Proc. VLDB*, 12(7), 2019.

[34] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proc. FPCA*, Nancy, France, September 1985.

[35] A. Kohn, V. Leis, and T. Neumann. Adaptive Execution of Compiled Queries. In *Proc. ICDE*, Paris, France, April 2018.

[36] C. Lattner, J.A. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR (arXiv)*, abs/2002.11054, 2020.

[37] C. Lawson. How Functions can Wreck Performance. *The Oracle Magician*, IV(1), January 2005. http://www.oraclemagician.com/mag/magic9.pdf.

[38] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[39] *Microsoft SQL Server 2019 Documentation*. http://docs.microsoft.com/en-us/sql.

[40] *MySQL 8.0 Documentation*. http://dev.mysql.com/doc/.

[41] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB*, 4(9), August 2011.

[42] *Oracle 19c Documentation*. http://docs.oracle.com/.

[43] *Oracle 19c PL/SQL Documentation*. http://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls.

[44] 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.

[45] *PostgreSQL 11 Documentation*. http://www.postgresql.org/docs/11/.

[46] *PostgreSQL 11 PL/pgSQL Documentation*. http://www.postgresql.org/docs/11/plpgsql.html.

[47] K. Ramachandra and K. Park. BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid. *Proc. VLDB*, 12(12), August 2019.

[48] K. Ramachandra, K. Park, K.V. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham. Froid: Optimization of Imperative Programs in a Relational Database. *CoRR (arXiv)*, abs/1712.00498, 2017. Extended version of [49].

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

[50] L.A. Rowe and M. Stonebraker. The POSTGRES Data Model. In *Proc. VLDB*, Brighton, UK, September 1987.

[51] A. Sabry and M. Felleisen. Reasoning about Programs in Continuation-Passing Style. *ACM SIGPLAN Lisp Pointers*, 5(1), January 1992.

[52] T.K. Sellis. Multiple-Query Optimization. *ACM TODS*, 13(1), March 1998.

[53] V. Simhadri, K. Ramachandra, A. Chaitanya, R. Guravannavar, and S. Sudarshan. Decorrelation of User Defined Functions in Queries. In *Proc. ICDE*, Chicago, IL, USA, March 2014.

[54] J. Snelson and J. Melton. XQuery Update Facility 3.0. W3C Working Group Note 24, January 2017. `https://www.w3.org/TR/xquery-update-30/`.

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

[56] *SQLite 3 Documentation.* `http://sqlite.org/`.

[57] A. Ulrich and T. Grust. The Flatter, the Better (Query Compilation Based on the Flattening Transformation). In *Proc. SIGMOD*, Melbourne, Australia, May 2015.

[58] O. Waddell and R.K. Dybig. Fast and Effective Procedure Inlining. In *Proc. Int'l Symposium on Static Analysis*, Paris, France, September 1997.