PL/SQL Without the PL

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

ABSTRACT

We demonstrate a source-to-source compilation technique that can translate user-defined PL/SQL functions into plain SQL queries. These PL/SQL functions may feature arbitrarily complex control flow—iteration, in particular. From this imperative-style input code we derive equivalent recursive common table expressions, ready for execution on any relational SQL:1999 back-end. Principally due to the absence of PL/SQL↔SQL context switches, the output plain SQL code comes with substantial runtime savings. The demonstration embeds the compiler into an interactive setup that admits function editing while live re-compilation and visualization of intermediate code forms happens in the background.

ACM Reference Format:
Denis Hirn and Torsten Grust. 2020. PL/SQL Without the PL. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3318464.3384678

1 COMPILING PL/SQL TO PLAIN SQL

Compilers for programming languages know a bag of tricks that transform *tail recursion into iteration.* The aim is to save stack space at run time and trade costly function calls for simple control flow (*i.e.*, loops, ultimately realized via `GOTO`). For the language PL/SQL[1], however, we argue that it is worth to reconsider this strategy. In fact, we propose to turn around 180° and demonstrate that the opposing transformation *from iteration to tail recursion* can greatly benefit the compilation and evaluation of PL/SQL functions:

- We compile PL/SQL functions that feature—arbitrary, potentially complex—iterative control flow into plain SQL queries that build on the `WITH RECURSIVE` construct. This compilation yields significant run time savings.
- Since the generated code does not rely on PL/SQL at all, the transformation equips any RDBMS with PL/SQL capabilities as long as the system implements `WITH RECURSIVE` (this can be particularly interesting for RDBMSs that do not implement user-defined functions at all, *e.g.*, SQLite3).

The demonstration offers an in-depth look at the compilation technique and makes the performance advantage tangible using a live, interactive setup.

Complex computation close to the data. Unlike database-external languages, PL/SQL bears the promise of performing complex computation right inside the RDBMS kernel. The source data is processed at its storage site and may remain in its original tabular shape.

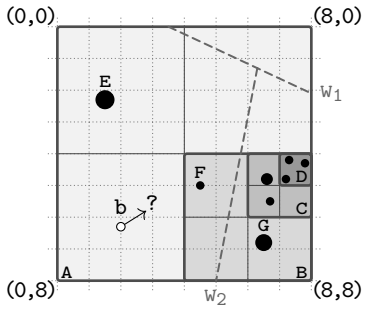


Figure 1: Plane with Barnes-Hut tree and walls w_i superimposed.

To exemplify, UDF `force` of Figure 2 is representative of the PL/SQL functions we can compile into plain SQL. `force(b,`θ`)` computes the gravitational force that the bodies populating a 2D plane exert on a new body `b` (see Figure 1). Bodies that are separated by any wall w_i do not affect each other. To forego the quadratic complexity of an algorithm that considers the force between each individual pair of bodies, `force` employs a *Barnes-Hut* tree [1] to organize the plane's body population into a hierarchy of ever smaller quadrants. Figure 3 shows tabular encodings of the Barnes-Hut tree and walls of Figure 1. The function descends the tree until it finds that the inner nodes approximate the aggregate center of mass of the bodies in a quadrant good enough (parameter $\theta \in [0, 1]$ controls the approximation's quality, with $\theta = 0$ requesting full accuracy).

While the full details of `force` are less relevant here, we note that the function's body features the constructs that characterize the imperative PL/SQL style of programming:

[1] The name *PL/SQL* has been coined by Oracle. This work extends to the language's variants known as *PL/pgSQL* (PostgreSQL) or *T-SQL* (MS SQL Server).

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3384678

```
1  CREATE FUNCTION force(b body, theta float) RETURNS point AS
2  $$
3   DECLARE
4    force      point := point(0,0); -- aggregated force on body
5    G CONSTANT float := 6.67e-11;   -- gravitational constant
6    Q          barneshut[];         -- Barnes-Hut nodes to visit
7    node       barneshut;           -- current Barnes-Hut node
8    children   barneshut[];         -- children of current node
9    dist       float;               -- distance and direction
10   dir        point;               --    between body and node
11   grav       point;               -- grav. effect of node on body
12  BEGIN
13   -- enter Barnes-Hut tree at the root
14   node = (SELECT t
15           FROM   barneshut AS t
16           WHERE  t.node = 0);                                    Q1
17   Q = array[node];
18   -- iterate while there are Barnes-Hut nodes to consider
19   WHILE cardinality(Q) > 0 LOOP
20    node = Q[1];
21    Q    = Q[2:];
22    dist = node.center<->b.pos;
23    dir  = node.center - b.pos;
24    grav = point(0,0);
25    -- bodies separated by walls do not affect each other          Q2
26    IF NOT EXISTS (SELECT 1
27                   FROM   walls AS w
28                   WHERE  (b.pos <= b.pos ## w.wall) <>
29                          (node.center <= node.center ## w.wall)) THEN
30     grav = (G * b.mass * node.mass / dist^2) * dir;
31    END IF;
32    -- Barnes-Hut optimization: approximate effect of distant bodies
33    IF (node.node IS NULL) OR (width(node.bbox) / dist < theta) THEN
34     force = force + grav;
35    ELSE
36     -- inspect area at higher resolution: descend into subtrees
37     children = (SELECT array_agg(t)
38                 FROM   barneshut AS t
39                 WHERE  t.parent = node.node);                     Q3
40     Q = Q || children;
41    END IF;
42   END LOOP;
43   -- return aggregated force on body
44   RETURN force;
45  END;
46 $$ LANGUAGE PLPGSQL STABLE STRICT;
```

Figure 2: Iterative UDF force, written in PostgreSQL's PL/pgSQL dialect ($Q_{1\ldots3}$: embedded SQL queries).

- iterative and conditional control flow (WHILE ··· LOOP, IF ··· THEN ··· ELSE) and blocks of sequential statements,
- embedded SQL queries ($Q_{1\ldots3}$) occuring in the role of expressions, as well as
- destructive update of variables. In the case of force, some of these variables have complex types (*e.g.*, row types, geometric types, or arrays).

The cost of PL/SQL↔SQL context switches. Despite its tight integration into the database kernel, PL/SQL performance often disappoints and it has become common developer wisdom to "avoid PL/SQL if possible" [6]. The evaluation of a query like SELECT force(b,0.5) FROM bodies AS b indeed leads to a constant back and forth between set-oriented SQL evaluation and iterative, statement-by-statement interpretation of PL/SQL. Invocation of force hands over control from SQL to the PL/SQL interpreter (which needs to start up or resume). On the first call to force, its embedded SQL queries $Q_{1\ldots3}$ are planned; on each subsequent call, these plans need to be re-instantiated. When the interpretation of force's body encounters Q_i, (1) PL/SQL is suspended, (2) SQL evaluation for Q_i's plan is performed, before (3) PL/SQL finally resumes. Iteration in SQL and PL/SQL—expressed in terms of SELECT ···FROM and WHILE···LOOP, respectively—only multiplies this effort. In fact, for PostgreSQL the bottom line shows that the system invests more than 35% of its time into PL/SQL↔SQL context switching [4].

barneshut

node	bbox	parent	mass	center	
A	box((0.0,0.0),(8.0,8.0))	□	16	(5.0,4.5)	A }
B	box((4.0,4.0),(8.0,8.0))	A	11	(6.6,5.5)	B } inner
C	box((6.0,4.0),(8.0,6.0))	B	6	(7.0,4.7)	C } nodes
D	box((7.0,4.0),(8.0,5.0))	C	3	(7.4,4.4)	D }
□	box((1.5,2.3),(1.5,2.3))	A	5	(1.5,2.3)	E }
□	box((4.5,5.0),(4.5,5.0))	B	1	(4.5,5.0)	F }
□	box((6.5,6.8),(6.5,6.8))	B	4	(6.5,6.8)	G }
□	box((6.6,4.8),(6.6,4.8))	C	2	(6.6,4.8)	H } bodies
□	box((6.7,5.5),(6.7,5.5))	C	1	(6.7,5.5)	I }
□	box((7.2,4.8),(7.2,4.8))	D	1	(7.2,4.8)	J }
□	box((7.3,4.2),(7.3,4.2))	D	1	(7.3,4.2)	K }
□	box((7.8,4.3),(7.8,4.3))	D	1	(7.8,4.3)	L }

walls

wall	
lseg((3.5,0.0),(8.0,2.1))	w_1
lseg((5.0,8.0),(6.3,1.3))	w_2

Figure 3: Tabular encodings of the Barnes-Hut quad tree, bodies, and walls in Figure 1 (□ represents NULL).

PL/SQL without the PL. The primary goal of this work is to entirely remove this costly friction between PL/SQL and SQL. Given a PL/SQL function f, we compile its body into a plain SQL query Q_f. If f contains iterative control flow, Q_f will be recursive (and thus use SQL:1999's standard WITH RECURSIVE clause). Inlining Q_f at the call sites of f yields pure SQL code: the original PL/SQL function f will not be invoked at run time (and has become obsolete, in fact). The system is able to compile and plan the resulting SQL-only query once—the repeated instantiation of embedded query plans (recall the Q_i) is avoided. At plan execution time, the RDBMS stays in its set-oriented query evaluation mode throughout, no statement-by-statement interpretation is required.

Contemporary RDBMSs reward this compilation effort with substantial performance improvements. It is common to observe run times drop by 50% or more if we evaluate function-heavy queries (see Section 2 and [4]).

2 FROM ITERATION TO TAIL RECURSION

The present research leans on compilation techniques developed by the programming languages community, even if we apply established strategies "backwards," as outlined below.

Compilation is performed in a pipeline of stages that receive a PL/SQL function f as input and ultimately emit plain

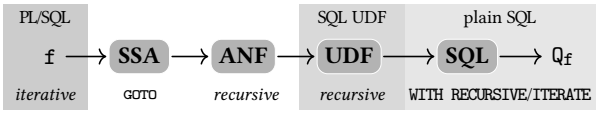


Figure 4: From PL/SQL to plain SQL: compilation stages.

SQL query Q_f ready to be inlined at `f`'s call sites. Refer to Figure 4 for an overview of these stages and the intermediate query forms they consume and produce. The demonstration makes all of these intermediate forms available for inspection in textual and/or graphical form (see Section 3).

From PL/SQL to GOTO. We begin by lowering the body of `f` into *static single assignment* (SSA) form [3]. SSA preserves the code's imperative flavor but imposes restrictions on variable assignment and usage that facilitate data flow analysis and a range of code simplifications. Expressions in these SSA programs are regular SQL expressions (like the embedded Q_i). Importantly, any PL/SQL iterative control flow—*e.g.*, `LOOP`, `EXIT`, `CONTINUE`, `FOR`, `WHILE`—is mapped to SSA's `GOTO` primitive. The resulting uniformity aids subsequent translation steps.

Note that the treatment of loops is a pivotal difference to the related *Froid* effort in MS SQL Server [6]. *Froid* compiles PL/SQL conditionals and sequential statements into plain SQL but will not admit any iterative control flow, arguably a staple of the imperative programming paradigm. We thus cover a significantly larger family of PL/SQL UDFs.

From GOTO to tail recursion. Unlike compilers for (functional) programming languages that pursue the reverse transformation, we bring the imperative SSA program into the restricted *administrative normal form* (ANF) [2]. Occurrences of `GOTO` are replaced by tail-recursive function calls. We obtain a family of mutually tail-recursive pure functions whose bodies are already free of any PL/SQL constructs. These functions could, in principle, be straightforwardly transformed into plain SQL UDFs. We do *not* follow this strategy here to avoid its substantial function call overhead.

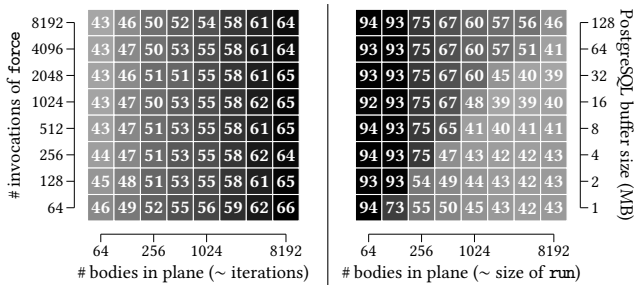
From tail recursion to WITH RECURSIVE. Instead, these ANF functions are merged to yield a single body of code. Inside that body, we locate non-recursive base cases as well as the sites of tail calls and use these to complete the holes in the generic SQL code template Q_f of Figure 5 [4]. In a nutshell, Q_f uses a recursive common table expression (CTE) to populate table `run` whose rows track the evolving state of `f`'s variables (also see Section 3). Since all recursive calls are tail calls, the fixpoint computation performed by the CTE [7] faithfully matches the semantics of the original `f` [4]. Q_f exposes the innards of the "black PL/SQL box" `f` and gives the RDBMS the opportunity jointly optimize Q_f (and the Q_i it contains) with the residual SQL query.

```
1 WITH {ITERATE|RECURSIVE} run("call"?, ..., force, Q, ..., result) AS (
2     ⟨initialize variable states⟩                    Qinit
3   UNION ALL
4     ⟨compute variable state updates⟩                Qrec
5 )
6 SELECT r.result
7 FROM   run AS r
8 WHERE  NOT r."call?";
```

Figure 5: Template for Q_f. With the holes Q_{init} and Q_{rec} filled, this SQL code can be inlined at `f`'s call sites.

Compiling PL/SQL away is worth the effort. For function `force`, experiments on PostgreSQL 11.3 manifest that Q_{force} only requires 43% to 66% of the run time of the PL/SQL variant (see Figure 6a where we vary the number of function invocations as well as intra-function `WHILE` loop iterations and thus control how many PL/SQL↔SQL context switches occur). We observed similar substantial improvements for a wide range of iterative PL/SQL functions [4].

Space-efficient tail recursion: WITH ITERATE. The more iterations `f` performs, the larger the `run` working table built by the recursive CTE in Q_f becomes. The darker region in Figure 6a documents how this space burden affects the run time of Q_{force}. Recall that `run` contains a trace of the state changes of `f`'s variables. Any iteration (or equivalently, evaluation of Q_{rec}, cf. Figure 5) will only consult the *most recent* variable states, however. An adaptation of `WITH RECURSIVE` that only keeps the working table's most recent row will thus suffice—all other rows may be immediately discarded. Earlier work [5] has coined the apt name `WITH ITERATE` for this simple variant of recursive CTEs. We built `WITH ITERATE` into PostgreSQL 11.3 and indeed observe the expected *additional* performance improvement over vanilla `WITH RECURSIVE` (see Figure 6b). The effect is particularly pronounced when many iterations are performed in scenarios of scarce buffer space



(a) Run time ratio Q_f/f (in %), # context switches varies.

(b) Impact of `WITH ITERATE` (ratio `ITERATE`/`RECURSIVE` in %).

Figure 6: Run time ratios (in %) on PostgreSQL 11.3. Light colors indicate an advantage for compiled code.

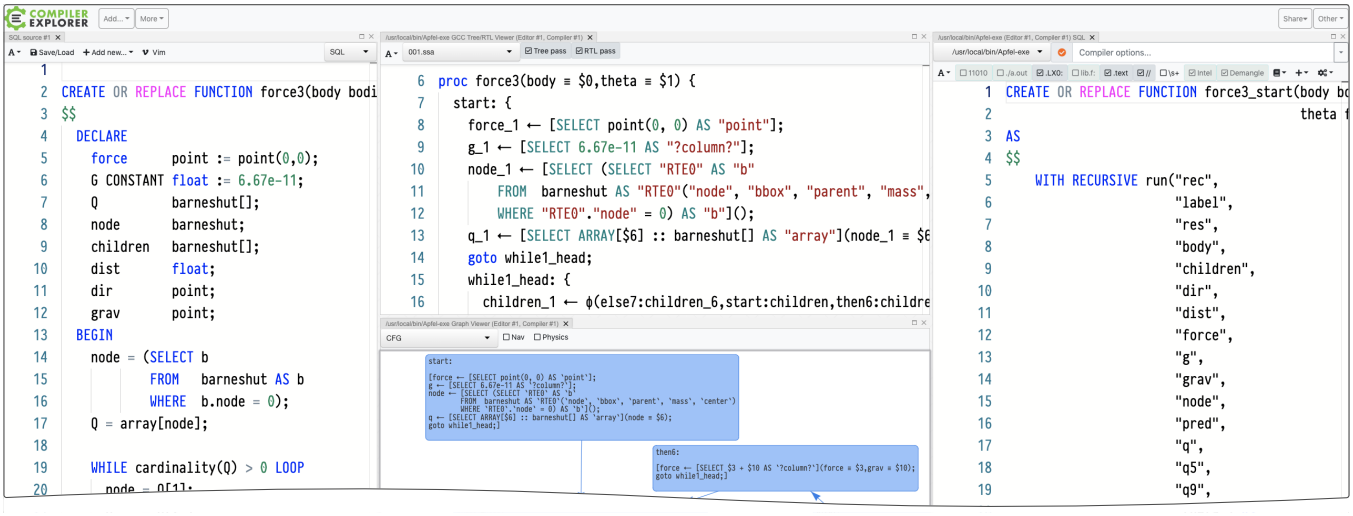


Figure 7: PL/SQL functions may be edited while all intermediate forms and the final SQL code are updated live.

(lower right region of Figure 6b): `WITH ITERATE` allocates literally no working memory and thus fully delivers on the space efficiency promise of tail recursion.

3 DEMONSTRATION SETUP

The compiler needs about 250 ms to translate a PL/SQL function into its plain SQL variant. This enables an interactive demonstration setup in which a PL/SQL function can be edited while it is compiled in the background and its SQL equivalent is displayed live. We have built such a tight editor/-compiler integration based on the widely known *Compiler Explorer* (normally hosted at `godbolt.org`). Output windows showing the pretty-printed intermediate forms—SSA, ANF, plain SQL UDFs, recursive CTE—as well as diagrammatic renderings thereof (*e.g.*, the function's control flow graph from which the SSA form is derived) update live and can be arranged at will. The screenshot of Figure 7 shows this PL/SQL-specific *Compiler Explorer*.

run

force	Q
(0,0)	{A}
(0,0)	{B,E}
(0,0)	{E,C,F,G}
(0,0)	{C,F,G}
(0,0)	{F,G,D,H,I}
(0,0)	{G,D,H,I}
$(-2.4^{-10}, 7.3^{-11})$	{D,H,I}
$(-2.5^{-10}, -2.3^{-11})$	{H,I}
$(-2.8^{-10}, -8.4^{-11})$	{I}
$(-3.1^{-10}, -1.2^{-10})$	{}
$(-3.1^{-10}, -1.2^{-10})$	{}

Figure 8: Traces disclose the workings of functions.

This in-depth compile-time account is complemented by facilities that help to understand the compiled function's run-time behavior. Much like a single-stepping debugger can provide, we output `run` tables that contain a full history of variable state changes that explain how the computation performed by the function body progressed. To illustrate, the absence of nodes J, K, L in column `Q` of Figure 8 (≡ variable `Q` of Figure 2) indicates that the Barnes-Hut optimization kept function `force` from inspecting these leaf nodes of the quad tree (cf. Figure 3).

Since we pursue source-to-source compilation, changes to the underlying RDBMS backend are not required. Code generation can indeed be adapted to target Oracle, for example [4]. The live demonstration is based on PostgreSQL 11.3 (but any recent version will do). Since we emit plain SQL, a rich dialect of PL/SQL functions can be compiled to also run on SQLite3, a system that does not natively support UDFs at all.

We further bring our PostgreSQL kernel that implements `WITH ITERATE`. The on-site audience will be able to experience the additional speed-up that comes with space-efficient iteration.

REFERENCES

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

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

[3] 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.

[4] C. Duta, D. Hirn, and T. Grust. Compiling PL/SQL Away. In *Proc. CIDR*, 2020.

[5] 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*, 2017.

[6] 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.

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