Snakes on a Plan

Compiling Python Functions into Plain SQL Queries

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

ABSTRACT

"Move your computation close to the data" is decades-old advice that is hard to follow if your code exhibits complex control flow. The runtime of such applications suffers from a continual back and forth between database-external code execution and plan-based SQL evaluation. We demonstrate the *ByePy* compiler which translates entire Python functions with arbitrary control flow—including deeply nested iteration—into plain recursive SQL:1999 queries. The invocation of a ByePy-compiled function enters the database engine once to execute the plan of a single query. Computation does not get much closer to the data than this. The system rewards this translation effort from Python to SQL with runtime improvements of up to an order of magnitude.

CCS CONCEPTS

• **Information systems** → **Structured Query Language**; • **Software and its engineering** → *Imperative languages*; *Recursion.*

KEYWORDS

Python, SQL, user-defined functions, compilation, recursion

ACM Reference Format:
Tim Fischer, Denis Hirn, and Torsten Grust. 2022. Snakes on a Plan: Compiling Python Functions into Plain SQL Queries. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3514221.3520175

1 COMPUTATION (NOT QUITE) CLOSE TO THE DATA

Applications over database-resident tables are well-advised *to perform their computation as close to the data as possible* [14]. Expressing computation in terms of queries over these tables

- leverages the carefully engineered data processing capabilities of modern database kernels, and
- avoids the extraction of sizable table contents which may quickly become stale and can overwhelm the application's heap space in the first place.

```
# pack all items of order into (several) packages of given max capacity
@to_compile
def pack(orderkey: int, capacity: int) -> list[list[int]]:
  # number of items in order
  n: int = SQL("""
               SELECT COUNT(*) FROM lineitem AS l WHERE l.l_orderkey = $1     Q1[·]
               """,
               [orderkey])
  # bail out if order not found or packages too small for largest item
  if n == 0 or
     capacity < SQL("""
                    SELECT MAX(p.p_size)                                      Q2[·]
                    FROM   lineitem AS l, part AS p
                    WHERE  (l.l_orderkey, l_partkey) = ($1, p.p_partkey)
                    """,
                    [orderkey]):
    return []
  packs: list[list[int]] = []      # start with an empty list of packages
  items: int = (1<<n) - 1          # full set of linenumbers {1,2,..,n}
  # iterate while there are more items to pack
  while items != 0:
    max_subset = 0                 # best subset of items (∅)...
    max_size   = 0                 #  ...and its size (0) so far
    subset     = items & -items    # start with single-item subset
    # iterate to find the largest subset of items that fits the package
    while True:
      # overall item size in current subset
      size: int = SQL("""
                      SELECT SUM(p.p_size)                                    Q3[·,·]
                      FROM   lineitem AS l, part AS p
                      WHERE  (l.l_orderkey, l_partkey) = ($1, p.p_partkey)
                      AND    $2 & (1 << (l.l_line_number - 1)) <> 0
                      """,
                      [orderkey, subset])
      # found a larger item subset that fits the package size?
      if size <= capacity and size > max_size:
        max_subset = subset                  # yes, remember the subset
        max_size   = size
      # considered all item subsets already?
      if subset == items:
        break                                # yes, done with this package
      else
        subset = items & (subset - items)    # no, consider next item subset
    # pack found subset of items (convert bit set to linenumber list)
    pack: list[int] = []
    for linenumber in range(n):
      pack.append(linenumber + 1 if max_subset & (1 << linenumber) != 0 else 0)
    # append this package of items to the list of packages
    packs.append(pack)
    # remove already packed items, process the rest
    items &= ~max_subset
  return packs
```

Figure 1: Original Python 3 source of function pack(o,c).

Following this decades-old advice all the way is often difficult, however: should the computation be complex, applications implement it in terms of—imperative, mostly—regular programs. Program execution is performed outside the database kernel, *i.e.*, far from the tabular data. Such programs are then tied to the data in terms of embedded SQL queries which identify and extract the data portions relevant for the next computation step.

To illustrate this style of data-intensive programming, consider the Python function `pack` of Figure 1 that operates over TPC-H tables `lineitem` and `part` [15]. Invoking `pack(`o`,`c`)` determines how the line items of order o can be packed into a set of boxes (all of which have capacity c). Given the TPC-H excerpt of Figure 2,

lineitem

l_orderkey	l_linenumber	l_partkey	⋯
o_1	1	p_1	
o_1	2	p_2	
o_1	3	p_3	
o_1	4	p_2	

parts

p_partkey	p_size	⋯
p_1	4	
p_2	2	
p_3	9	

Figure 2: TPC-H tables `lineitem` and `parts` (excerpt only).

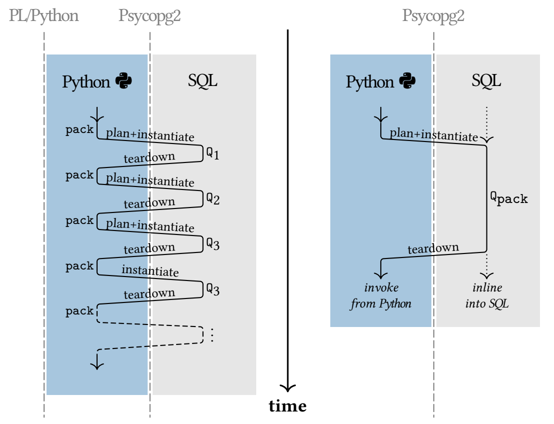
`pack(`o_1`,10)` returns the nested array `[[0,0,3,0],[1,2,0,4]]` of line item numbers, indicating that the third item (part p_3 of size 9) is to be placed inside the first box, while the remaining three items (of aggregate size $4 + 2 + 2 < 10$) fit into a second box .

Function `pack` follows a simple greedy strategy to fill the boxes. This approach is moderately complex but `pack`'s code already exhibits quite intricate nesting of `while` and `for` iteration with early exits (`break`), conditional execution (`if`/`else`), statement sequencing, and variable references/updates, all of which are signature constructs of the imperative programming paradigm. The function embeds SQL queries—issued via `SQL(`*query text*`,[`*parameters*`])`—to check line item count and size of the largest item ($\mathtt{Q}_1$ and $\mathtt{Q}_2$) and to determine the overall volume of a subset of items ($\mathtt{Q}_3$). Note that $\mathtt{Q}_3$ is located inside the innermost `while` loop and is thus evaluated repeatedly while `pack` considers all promising item subsets.

Countless database APIs and libraries promote this mixture of imperative and query-based programming. In this demonstration, we focus on Python 3 and its library Psycopg2, a widely deployed combination targeting PostgreSQL backends [10, 11]. Equivalents exist for practically any programming language and database backend.

One function, dozens of plans. The runtime performance of applications based on `pack`-like functions often is sobering and may even turn out to be impractical [13]. Each invocation of `pack` leads to a constant back and forth between the Python interpreter and the SQL database engine:

1. Execution is initiated by the Python side which follows the function body's control flow and interprets statements.



(a) Python code for `pack` invokes SQL queries $\mathtt{Q}_i$. **(b) With function `pack` compiled to pure SQL.**

Figure 3: Interplay of Python interpreter and SQL engine. To the right of boundary ┆, we are inside the DBMS process.

2. When embedded query `SQL(`$\mathtt{Q}_i$`,[`$v_1,\ldots,v_n$`])` is encountered, the database engine plans query $\mathtt{Q}_i$ (or retrieves a formerly constructed plan from a cache), instantiates the plan with parameters $v_1, \ldots, v_n$, evaluates the plan, then tears down transient plan data structures, before it hands the result back to Python.

Figure 3a depicts how we continually cross the process boundary (marked ┆) between Python and the database kernel. The system thus repeatedly pays the price for the planning and teardown of the $\mathtt{Q}_i$—the latter is particularly significant if queries are evaluated iteratively (recall $\mathtt{Q}_3$ in `pack`). Note that this dire situation only marginally improves if we host the Python interpreter inside the database kernel, *e.g.*, as implemented in PostgreSQL's PL/Python [10, § 46]: the process boundary moves but the piecemeal plan evaluation remains. For `pack`, we find that one function invocation typically leads to 40+ SQL queries being planned and evaluated. The resulting friction at run time is substantial and, indeed, it is established developer lore that complex computation is better hosted outside the database system [13].

2 SNAKES 🐍 ON A PLAN

We demonstrate *ByePy*, a compiler that translates a Python function, f say, into a *single plain SQL:1999 query* $\mathtt{Q}_f$. Query $\mathtt{Q}_f$ embodies the *entire computation* originally performed by the Python code in f. If we invoke $\mathtt{Q}_f$ (instead of f) on the Python side, we cross the boundary into the database kernel once: a *single plan* for $\mathtt{Q}_f$ is created, instantiated, evaluated, and ultimately torn down. We only return to Python once the final function result is available. Figure 3b depicts how SQL query $\mathtt{Q}_{\mathtt{pack}}$ assumes the former role of Python function `pack` (follow the path ↰, annotated *invoke from Python*). The constant back and forth between the external interpreter and the database engine is avoided.

This compilation from Python to pure SQL yields substantial runtime reductions. The effect multiplies if the invocations of f lie on a "hot", repeatedly executed code path as is typical for data-intensive applications. Figure 4 records the wall clock time of experimental runs in which function `pack(`o`,60)` of Figure 1 was evaluated over (a growing) fraction of the finished orders o of a TPC-H database. Processing all such orders in a 1 GB TPC-H instance leads to 729 413 invocations of `pack`. These, in turn, lead the Python implementations of the function to execute the embedded SQL queries $\mathtt{Q}_i$ 32 587 763 times in total: PL/Python thus pays the price for plan instantiation, execution, and teardown more than 32 million times. Just as often, Psycopg2 additionally incurs the penalty of switching between the external interpreter and the database kernel. At the bottom line, the Python variants of `pack` clock in at 1.67 and 3.63 ms per invocation. (All measurements performed with Python 3.8.10 and PostgreSQL 11.3.)

Invoking the ByePy-compiled pure SQL variant of `pack` enters the database kernel *once* per invocation to instantiate and evaluate the plan for $\mathtt{Q}_{\mathtt{pack}}$ (this plan is more complex than those for $\mathtt{Q}_{1\_3}$, but it, too, remains in the engine's session-wide plan cache). We have measured `pack`'s SQL version at 0.72 ms per invocation, using only 1/5 of the runtime required by the Python original it was derived from. We have, indeed, observed *runtime reductions about an order of magnitude*, in particular for Python functions that iterate the evaluation of embedded SQL code.

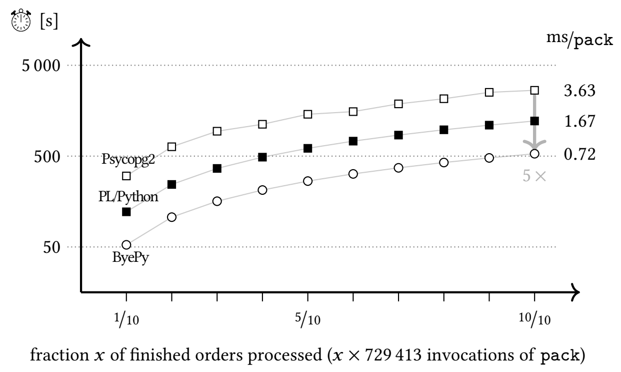


Figure 4: Runtime of pack before/after compilation to SQL.

From Python to pure SQL. The ByePy compiler emits a SQL query Q_f that

- contains f's embedded queries Q_i as subqueries and, most importantly,
- realizes Python's imperative constructs, including statement sequences, arbitrary control flow, or updateable variables.

ByePy maps the rich semantics of these programming language constructs onto a *recursive common table expression* (CTE, as introduced by SQL:1999 in terms of WITH RECURSIVE) [6]:

```
WITH RECURSIVE T(···) AS
  (Qstart UNION ALL Qrec(T))
TABLE T;
```

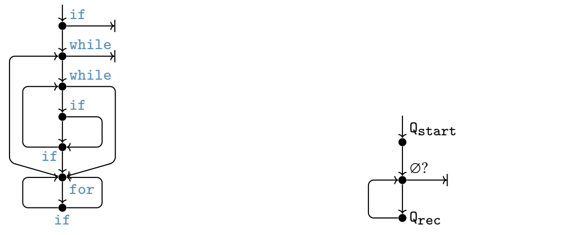
The present demonstration has been built on top of PostgreSQL [10], but recursive CTEs enjoy widespread support in off-the-shelf database systems, including Oracle, MS SQL Server, MySQL, or SQLite. Any such system would be a suitable target database platform for the ByePy compiler: support for plain SQL:1999 suffices. In particular, neither PL/Python, PL/SQL, or similar are involved or required.

Python subset understood by ByePy. We have built ByePy with a focus on the Python language features that characterize the imperative programming style many developers are well versed in. Supported Python constructs currently include

- looping and iteration (while, for v in range/array),
- flow control statements (continue, break, return),
- conditional statements (if, elif, else) as well as expressions (e_1 if e_2 else e_3),
- variable assignment ($v = e$, v += e, $v[e_1] = e_2$) and reference,
- lists ($[e_1, \dots, e_n]$), indexed access and slicing ($e_1[e_2]$, $e_1[e_2:e_3]$), stateful list methods (e.pop, e.append, e.extend),
- a large range of builtin operators and functions (+, %, **, &, ~, <<, <=, ==, and, ..., len, max, ceil, sqrt, coalesce, ...), and
- embedded read-only queries (SQL(q, $[e_1, \dots, e_n]$)).

Besides the obvious mapping of values of the atomic Python types int, float, bool, str to SQL, ByePy supports advanced typing aspects of contemporary Python, *e.g.*, type annotations ($e : \tau$), Python dataclasses (which are compiled into SQL composite row types), and Python literals (translated into SQL enumeration types) [12].

The demonstration setup features a wide variety of Python functions chosen to exercise all of the above language features.



(a) Nested control flow in Python function pack. **(b) Single loop performed by a SQL:1999 recursive CTE.**

Figure 5: ByePy maps complex control flow to a single loop.

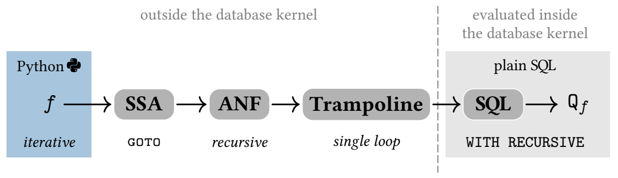


Figure 6: Compiler stages and intermediate program forms.

Arbitrary control flow into a single loop. There is a considerable gap between the arbitrary—often iterative and deeply nested—control flow that is typical for Python code and the fixed-point semantics embodied by SQL's recursive CTE [1]. To see this, compare the control flow graph of Python function pack shown in Figure 5a with the simple single-loop computation performed by a recursive CTE (Figure 5b): in the zeroth loop iteration, the CTE executes initial query Q_{start} once. Q_{rec} then is repeatedly evaluated over the results of the previous iteration. Intermediate results are collected and ultimately returned when an iteration yields no rows.

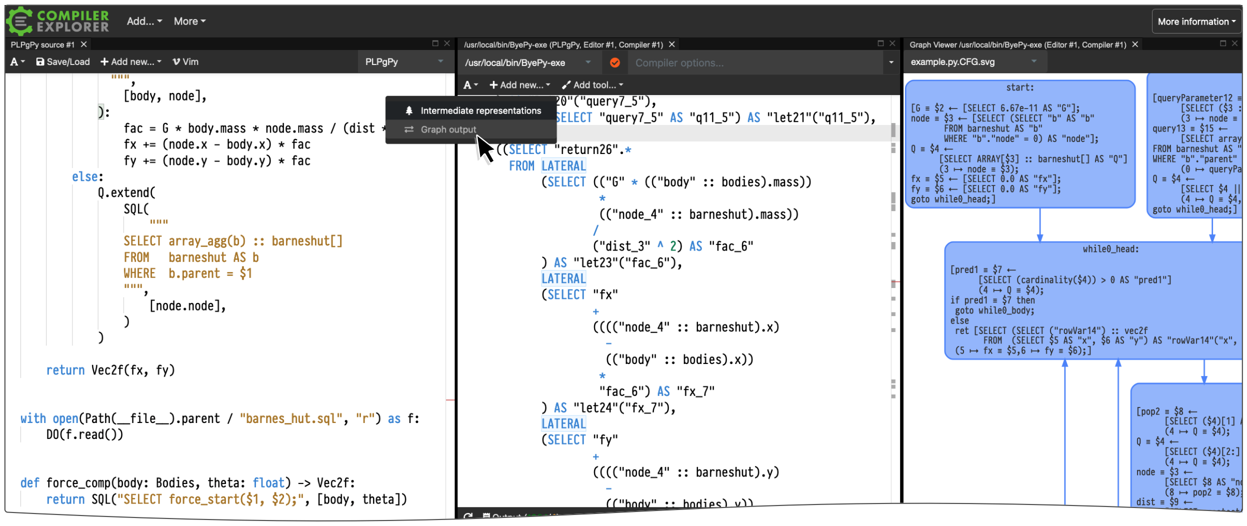
ByePy builds on compilation techniques established by the programming languages community to bridge this chasm. We adapt these techniques and arrange them in a pipeline to transform the imperative Python function f into a declarative SQL query Q_f. The compiler is characterized by the intermediate program forms that f goes through (also see Figure 6):

1. Bring f into SSA form [4] in which iterative as well as conditional control flow is exclusively expressed in terms of GOTO,
2. translate the resulting graph of SSA blocks into a bundle of tail-recursive functions in ANF [2],
3. form a central *trampoline* function [7] which dispatches to the functions in the bundle, then loops back to itself, and
4. inline the functions into the trampoline, after which the recursive CTE Q_f can be read off this final intermediate form.

Since this compilation chain defines the ByePy project, we have ensured that the live demonstration exposes f in its various intermediate forms for inspection and experimentation (see Section 3).

We have implemented ByePy as a standalone compiler that reads Python application source and emits SQL code for all Python functions that carry a @to_compile decorator (see Line 2 in Figure 1). A complete application may then comprise of such compiled (presumably, data-intensive) functions and regular interpreted Python code which remains outside the database.

ByePy builds on earlier work to compile user-defined PL/SQL functions (UDFs) into plain SQL queries. This includes the seminal



```
                """,
                [body, node],
            ):
                fac = G * body.mass * node.mass / (dist *
                fx += (node.x - body.x) * fac
                fy += (node.y - body.y) * fac
        else:
            Q.extend(
                SQL(
                    """
                SELECT array_agg(b) :: barneshut[]
                FROM   barneshut AS b
                WHERE  b.parent = $1
                """,
                    [node.node],
                )
            )

    return Vec2f(fx, fy)


with open(Path(__file__).parent / "barnes_hut.sql", "r") as f:
    DO(f.read())


def force_comp(body: Bodies, theta: float) -> Vec2f:
    return SQL("SELECT force_start($1, $2);", [body, theta])
```

```
20"("query7_5"),
SELECT "query7_5" AS "q11_5") AS "let21"("q11_5"),

((SELECT "return26".*
  FROM LATERAL
       (SELECT (("G" * (("body" :: bodies).mass))
                 *
                 (("node_4" :: barneshut).mass))
                /
                ("dist_3" ^ 2) AS "fac_6"
       ) AS "let23"("fac_6"),
       LATERAL
       (SELECT "fx"
                +
                (((("node_4" :: barneshut).x)
                  -
                  (("body" :: bodies).x))
                 *
                 "fac_6") AS "fx_7"
       ) AS "let24"("fx_7"),
       LATERAL
       (SELECT "fy"
                +
                (((("node_4" :: barneshut).y)
                  -
                  (("body" :: bodies).y))
```

```
start:

[G ≡ $2 ⟵ [SELECT 6.67e-11 AS "G"];
node ≡ $3 ⟵ [SELECT (SELECT "b" AS "b"
       FROM barneshut AS "b"
       WHERE "b"."node" = 0) AS "node"];
Q ≡ $4 ⟵
      [SELECT ARRAY[$3] :: barneshut[] AS "Q"]
      (3 ↦ node ≡ $3);
fx ≡ $5 ⟵ [SELECT 0.0 AS "fx"];
fy ≡ $6 ⟵ [SELECT 0.0 AS "fy"];
goto while0_head;]
```

```
[queryParameter12 ≡
      [SELECT ($3 :
      (3 ↦ node ≡
query13 ≡ $15 ⟵
      [SELECT array
FROM barneshut AS "
WHERE "b"."parent"
      (0 ↦ queryPa
Q ≡ $4 ⟵
      [SELECT $4 ||
      (4 ↦ Q ≡ $4,
goto while0_head;]
```

```
while0_head:

[pred1 ≡ $7 ⟵
      [SELECT (cardinality($4)) > 0 AS "pred1"]
      (4 ↦ Q ≡ $4);
if pred1 ≡ $7 then
 goto while0_body;
else
 ret [SELECT (SELECT ("rowVar14") :: vec2f
        FROM  (SELECT $5 AS "x", $6 AS "y") AS "rowVar14"("x",
 (5 ↦ fx ≡ $5,6 ↦ fy ≡ $6);]
```

```
[pop2 ≡ $8 ⟵
      [SELECT ($4)[1]
      (4 ↦ Q ≡ $4);
Q ≡ $4 ⟵
      [SELECT ($4)[2:]
      (4 ↦ Q ≡ $4);
node ≡ $3 ⟵
      [SELECT $8 AS "n
      (8 ↦ pop2 ≡ $8);
dist ≡ $9 ⟵
```

Figure 7: Python source edits (left) lead to live updates of generated SQL code (center) and intermediate program forms (right).

work on *Froid* [13]—which revived a now very active branch of research—as well as our own efforts [8, 9]. That work particularly addressed the divergence between the statement-by-statement interpretation of PL/SQL UDFs and the set-oriented and plan-based evaluation of declarative queries. While PL/SQL and SQL are both native languages to the database, back then we already conjectured that database-external application code could be subject to compilation to plain SQL, too [5]. ByePy is our first embodiment of this idea. The compiler embraces a Python subset that significantly extends the dialects admitted by earlier efforts [16] and translates arbitrary control flow beyond *folds* (unlike [13]) practically instantly (unlike the synthesis-based [3, 17]).

3 A LIVE DEMONSTRATION OF ByePy

ByePy translates a screenful of Python code within fractions of a second. We build on this compilation speed to create an interactive demonstration setup for ByePy (Figure 7) in which edits of the Python source lead to live updates of all outputs produced by the individual compiler stages. This demonstrator is derived from the browser-based *Compiler Explorer* (`godbolt.org`) that we have adapted to accommodate the intermediate program forms used by ByePy (recall Figure 6):

- We visualize the (potentially complex) control flow of the Python input function f to make the challenge of compiling towards the single-loop CTE form tangible.
- We offer textual and graphical representations of SSA block structure or ANF call graphs. Both reveal how Python constructs are expressed in terms of `GOTO` or mutual tail recursion, respectively.
- We prepare a formatted display of the SQL code $\mathtt{Q}_f$ emitted by ByePy. Seeing f and $\mathtt{Q}_f$ side by side helps to understand how the Python semantics is reflected in terms of (recursive) SQL.

Both, input and emitted code, are executable to demonstrate first-hand that $\mathtt{Q}_f$ can indeed assume the role of f. To facilitate experimentation with the generated code, we wrap $\mathtt{Q}_f$ in a plain SQL UDF that can be invoked from Python as well as directly within a SQL query. (Aside: Note that the latter demonstrates how Python could be used to develop bits of *SQL-only applications*. Indeed, we measured 0.26 ms per invocation if $\mathtt{Q}_{\mathtt{pack}}$ is inlined into an enclosing SQL query since we bypass any Python-induced latency—see the straight path ⇣ in Figure 3b.)

To jump-start the on-site demonstration, we will bring a collection of Python functions implementing algorithms from various domains over a variety of data types (the screenshot of Figure 7 shows an excerpt of inter-body gravity computation based on Barnes-Hut trees, for example).[1] These functions will be editable. Entirely new code can be constructed on a blank page at the audience's request.

REFERENCES

[1] F. Bancilhon. Naive Evaluation of Recursively Defined Relations. In *On Knowledge Base Management Systems*, pages 165–178. Springer, 1986.
[2] M. Chakravarty, G. Keller, and P. Zadarnowski. A Functional Perspective on SSA Optimisation Algorithms. *ENTCS*, 82(2), 2004.
[3] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing Database-Backed Applications with Query Synthesis. In *Proc. PLDI*, 2013.
[4] 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.
[5] C. Duta, D. Hirn, and T. Grust. Compiling PL/SQL Away. In *Proc. CIDR*, 2020.
[6] 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, 1996.
[7] S.E. Ganz, D.P. Friedman, and M. Wand. Trampolined Style. In *Proc. ICFP*, 1999.
[8] D. Hirn and T. Grust. PL/SQL Without the PL. In *Proc. SIGMOD*, 2020.
[9] D. Hirn and T. Grust. One `WITH RECURSIVE` is Worth Many `GOTO`s. In *Proc. SIGMOD*, 2021.
[10] *PostgreSQL version 11*. `https://www.postgresql.org/docs/11/`.
[11] *Psycopg2—PostgreSQL Adapter for Python*. `https://www.psycopg.org/docs/`.
[12] *Type Hints in Python 3*. `https://docs.python.org/3/library/typing.html`.
[13] 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.
[14] L.A. Rowe and M. Stonebraker. The POSTGRES Data Model. In *Proc. VLDB*, 1987.
[15] *TPC Benchmark H (Revision 3.0.0)*, 2021. `http://tpc.org/tpch`.
[16] K. Venkatesh Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proc. SIGMOD*, 2016.
[17] G. Zhang, Y. Xu, X. Shen, and I. Dillig. UDF to SQL Translation Through Compositional Lazy Inductive Synthesis. In *Proc. OOPSLA*, 2021.

[1]These Python functions as well as their recursive SQL equivalents can also be found at `https://github.com/ByePy/examples`.