To Iterate Is Human, to Recurse Is Divine — Mapping Iterative Python to Recursive SQL

Tim Fischer[1]

Abstract: Writing complex algorithms and iterative computations in SQL is difficult at best, commonly leading to code that intermingles looping control flow with database access. This yields programs with control flow that rapidly hops in and out of the database, with each roundtrip incurring significant overhead. We present the ByePy compiler, which can compile entire Python functions directly to plain recursive SQL:1999 queries. By doing so, the compilation eliminates all but a single roundtrip, leading to runtime speedups of up to an order of magnitude.

Keywords: SQL; Python; Compilation

1 Introduction

The performance of all applications stands and falls with the efficient use of resources, e.g., compute, memory, network, etc. In the realm of database-backed applications, developers can optimize the usage of many, if not most, of these resources by adhering to the decades-old mantra of database development: "*Move your computation close to the data*" [RS87]. To do so, developers need to express their computations *and* data in a form that databases can ingest and process. Nowadays, this usually means storing the data in tables and expressing the computations over it in terms of the ubiquitous SQL.

Following the mantra and moving *all* computation into the database is often difficult. The main stumbling block is the *impedance mismatch* between the *declarative* paradigm underlying SQL and the *imperative* paradigm most developers are more familiar with. This mismatch leads developers most comfortable with imperative programming to write programs that perform the bulk of their computation outside the database, i.e., far away from the data. Such programs are littered with intermittent database access throughout; consider the implementation of function `march` in the left half of Fig. 1 (`march` computes the outline of a 2D shape). During the execution of such a program, it will have to perform many complete round trips, i.e., the program's control flow has to move from Python to the database and back again. With each additional round trip incurring resource overhead.

Intending to reach code that behaves like the right side of Fig. 1, that is, minimizing round trips while retaining the imperative paradigm, Ramachandra et al. introduced Froid

[1] Eberhard Karls Universität Tübingen, Wilhelm Schickard Institut tim.fischer@uni-tuebingen.de

doi:10.18420/BTW2023-73

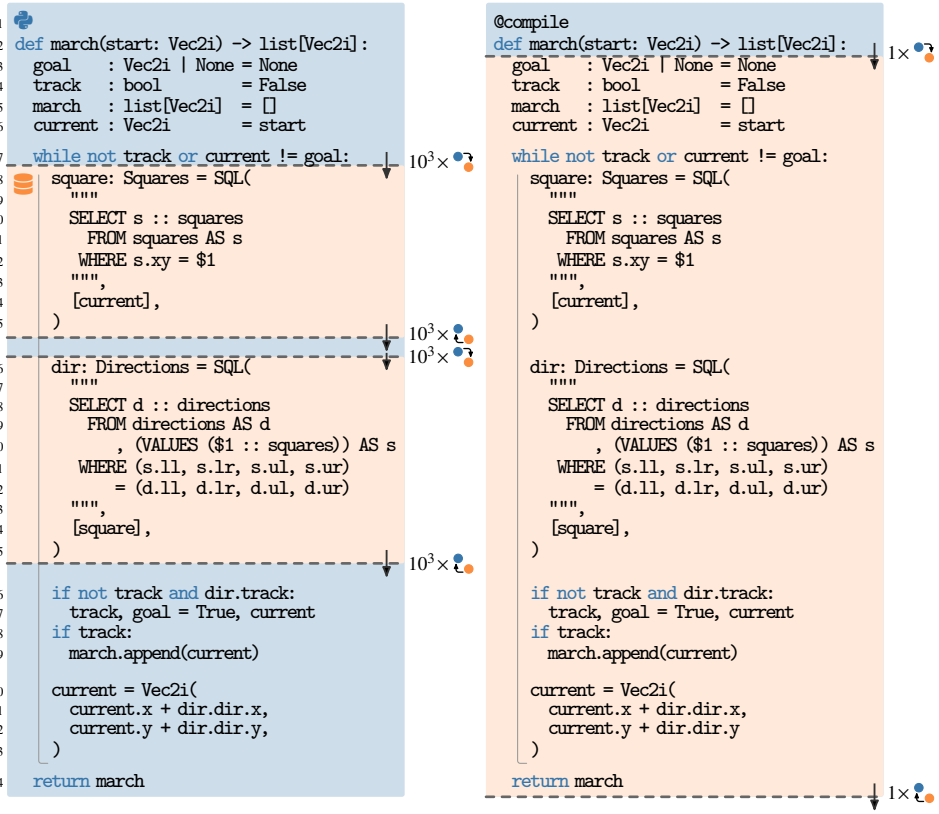


Fig. 1: Execution context boundaries between Python and the database in iterative code with embedded queries and code compiled with ByePy.

[Ra17]. Focussing on optimizing simple non-iterative PL/SQL functions by compiling them to SQL, Froid was restricted to code with linear control flow. Building on this idea, Hirn and Grust introduced a new compilation approach that extended the compilation to support non-linear control flow constructs, e.g., loops [HDG20; HG20; HG21]. In [FHG22], we demonstrated the applicability of this approach to languages other than PL/SQL via ByePy, a Python frontend for [HG21]. Since then, we have been working on extending the set of language constructs that ByePy can digest. In addition to the features presented in [FHG22], it now also supports the following:

- dictionaries with string-valued keys and "JSON-valued" entries,
- delete statements on containers (e.g., `del some_list[2:5]`),
- falsifiability of builtins (e.g, `while some_list: ...`),
- nested `None` disambiguation (e.g, `if ... and x is not None and ...: ...`), and
- arbitrarily nested (augmented-)assignment (e.g., `v[0].my_dict["key"] += 1`).

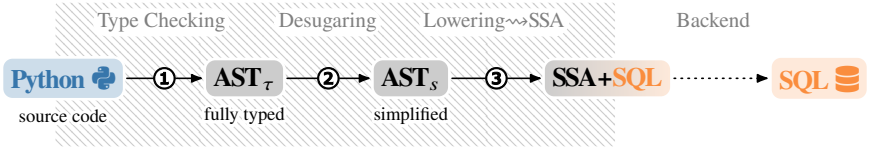


Fig. 2: Intermediate representations in the BYEPY frontend.

2 The BYEPY Compiler

The BYEPY compiler consists of two major parts. The first of which is the novel frontend which compiles Python to a mix of Static Single Assignment form (SSA) to represent the general control flow and SQL to represent embedded expressions. Furthermore, the second part consists of the SSA to SQL compilation pipeline elaborated in [HG21].

BYEPY focuses on computations over database-resident data; as such, we limit the supported language features to a subset most commonly used in conjunction with such computation—think conditionals, loops, flow control statements like `break`, and complex assignments like `v[0].att += 1`. The frontend wrangles Python programs using this subset into the SSA+SQL representation in three distinct stages, as depicted in Fig. 2, those being ① performing soundness and type checking, ② simplifying particularly complex statements and expressions, and ③ lowering the AST into the combined SSA+SQL representation.

① Type Checking Aside from the *conceptual* impedance mismatch between the imperative paradigm of Python and the declarative paradigm of SQL there is a *structural* impedance mismatch. Python is a dynamically typed language, meaning types are reified *at runtime*. SQL, on the other hand, is statically typed, where types are already explicitly declared *before runtime*. To bridge this gap, BYEPY implements a type-checking stage that enriches a minimally typed AST such that each expression is annotated with an appropriate type. Minimally typed refers to the fact that BYEPY requires type annotations in situations where the type inference does not have enough information—e.g., function parameters, variable declarations, or function return types.

② Desugaring Following the type checking, we rewrite parts of the AST in terms of simpler syntactic constructs. Doing so simplifies the subsequent steps greatly as it limits the amount of different syntactic constructs they are required to handle. These rewrites include the following:

- reducing the set of used operators by rewriting more complicated ones in terms of simpler one (e.g., `x not in y` ↦ `not (x in y)`),
- placing appropriate "casting expression" where Python's duck-typing would do so during runtime (e.g., `if some_list: ...` ↦ `if len(some_list) > 0: ...`),
- replacing stateful expressions with an equivalent series of assignments and variables, and
- rewriting of arbitrarily complex assignments into their simplest equivalents (e.g., `v[0]["key"] += 1` ↦ `v = [{**v[0], "key": v[0]["key"] + 1}] + v[1:]`).

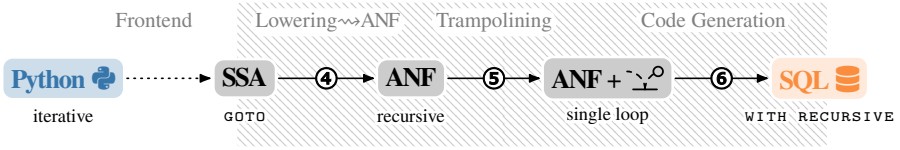


Fig. 3: Intermediate representations in the BytePy backend.

③ **Lowering⇝SSA** The desugaring produces an equivalent AST in which all *stateful computation* is expressed solely through *single variable assignments* and *statement-level control flow*. With such an AST in hand, we can finally apply the lowering. In short, we translate all control flow constructs into equivalent labeled blocks and `GOTO`s while translating the remainder directly to equivalent SQL queries.

Once the frontend has compiled all Python specifics away, the backend pipeline designed by Hirn and Grust comes into play. In short, it applies a series of three transformations depicted in Fig. 3 which results in an equivalent *recursive* SQL:1999 query, i.e., a recursive common table expression (CTE).

④ **Lowering⇝ANF** The control flow, which is expressed in terms of SSA, is lowered to Administrative Normal Form (ANF) using a transformation described by Chakravarty et al. in [CKZ04]. In short, we turn all blocks into functions, all `GOTO`s into calls of those functions, and all assignments into `LET`-expressions. Of particular note is that the lowering to ANF places the calls replacing the `GOTO`s in the tail position.

⑤ **Trampolining** Lowering to ANF, generally, leads to a family of recursive functions. To facilitate the compilation into the SQL-based CTE form, we subject this family of functions to the trampoline transformation [GFW99], which yields a single-loop computation that fits the CTE semantics.

⑥ **Code Generation** The last step is to generate SQL code equivalent to the program in trampolined ANF. We can do so by encoding `LET`-expressions as `LATERAL`-joins, recursive calls as `SELECT`-clauses containing the parameters, and conditional expressions as `UNION`s of the individual branches with mutually exclusive `WHERE`-clauses.

The generated query implements the trampoline through a recursive CTE in which each recursive step handles one trampoline transition. Thus, all program state resides within the working table; this includes both the state of the control flow and the bindings of the program's live variables. Each recursive step generates a new row representing the result of the transition, containing both new variable bindings and copies of the unchanged bindings. This behavior can lead to performance impacts for functions whose local variables carry sizable data structures—like long arrays. The right edge of Fig. 4a exemplifies this.

Tab. 1: A collection of Python functions with roundtrips before and speedup after compilation.

Function		CC	Loops	# per call	Runtime (Speedup) after compilation
`march`	track border of 2D object (Marching Squares)	5	Q,	2000	13% (7.6×)
`savings`	optimize supply chain of a TPC-H order	4	QQ, QQQQ,	18	5% (19.5×)
`packing`	pack TPC-H lineitems tightly into containers	9	QQ, Q, ,	45	16% (6.3×)
`force`	*n*-body simulation (Barnes-Hut quad tree)	5	Q, Q,	126	27% (3.9×)
`margin`	buy/sell TPC-H orders to maximize margin	5	Q, QQ,	61	24% (4.2×)
`markov`	Markov-chain based robot control	5	QQQ,	3000	39% (2.6×)
`vm-collatz`	calculate the *collatz conjecture* on a simple VM	17	Q,	67	30% (3.3×)
`vm-padovan`	calculate the *padovan sequence* on a simple VM	17	Q,	7100	12% (8.5×)

3 Experimenting with the *Divine*

We claim that compiling Python to SQL using the BytePy pipeline has the capability of speeding up programs quite drastically depending on their complexity. This section supports this claim through eight sample functions with varying complexity and quantifies how the compilation affects their runtimes. We performed all measurements with PostgreSQL 11.3 and Python 3.8 running on a 64-bit Linux x86 host (2× AMD EPYC™ CPUs at 2.8 GHz, 2 TB of DDR4 RAM). All presented results represent the median of five runs.

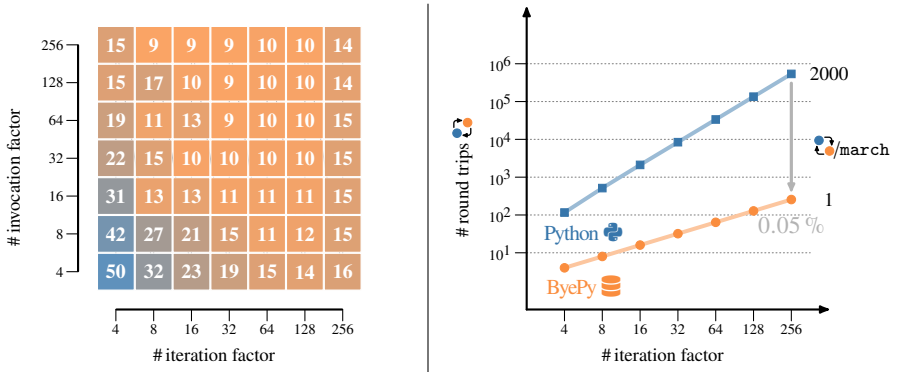
Tab. 1 lists the eight sample functions; you can find the original Python source code, compiled SQL queries, and appropriate data generators on *GitHub*[2]. They cover a wide range of use cases and classes algorithms, e.g., optimization problems over TPC-H data, simulation of VMs, or algorithms over 2D point data. The columns **CC** (cyclomatic complexity) and **Loops** give some insight into the structure of the functions without looking at the source code. Especially the latter gives a sense of where the embedded queries (`Q`) sit inside the functions control flow.

Zooming in on `march` in Fig. 4a, we can see that the performance of the compiled functions can be sensitive to the size of the data they operate on. In the lower-left corner, the planning required for the query outweighs to performance benefits introduced by the compilation. Fig. 4b shows us the expected effect on the number of roundtrips; increasing the number of iterations and invocations also increases the round trips a program encounters. Furthermore, we can also see that compilation significantly decreases the required round trips .

4 Wrapping Up

When working with database-resident data, most Python developers opt to perform complex computations outside of the database. Embedding database access in (potentially deeply nested) loops raises significant performance concerns. To minimize the resulting round

[2] `https://github.com/BytePy/examples`



(a) Runtime (in % of Python) after compilation

(b) Comparison of round trips along the diagonal of Fig. 4a

Fig. 4: Deeper analysis of the runtimes and round trips of the `march` function.

trips such code experiences during runtime, we developed BytePy; a Python frontend for the PL/SQL to SQL compilation pipeline described in [HG21]. The compilation yields runtime speedups of up to an order of magnitude on a wide range of functions, from optimization problems to stochastic processes.

Currently, we are in the process of introducing PostgreSQL's geometric types, functions, and operators as optional extensions to the BytePy dialect. Beyond these extensions, we plan to expand the set of Python language features BytePy supports. Extensions that appear to be in immediate reach include things like multi-variable assignments (e.g., `a, b = 1, 2`) and comprehensions (e.g., `[f(e) for e in some_list if p(e)]`). In the *not so near* future, we hope to integrate the compilation directly into the `@compile` decorator, enabling easier use of BytePy in the wild.

References

[CKZ04] Chakravarty, M. M.; Keller, G.; Zadarnowski, P.: A Functional Perspective on SSA Optimisation Algorithms. COCV'03/, 2004.

[FHG22] Fischer, T.; Hirn, D.; Grust, T.: Snakes on a Plan: Compiling Python Functions into Plain SQL Queries. In. SIGMOD '22, 2022.

[GFW99] Ganz, S. E.; Friedman, D. P.; Wand, M.: Trampolined Style. In. ICFP '99, 1999.

[HDG20] Hirn, D.; Duta, C.; Grust, T.: Compiling PL/SQL Away. In. CIDR '20, 2020.

[HG20] Hirn, D.; Grust, T.: PL/SQL Without the PL. In. SIGMOD '20, 2020.

[HG21] Hirn, D.; Grust, T.: One WITH RECURSIVE is Worth Many GOTOs. In. SIGMOD '21, 2021.

[Ra17] Ramachandra, K.; Park, K.; Emani, V.; Halverson, A.; Galindo-Legaria, C.; Cunningham, C.: Froid: Optimization of Imperative Programs in a Relational Database. In. VLDB '17, 2017.

[RS87] Rowe, L.; Stonebraker, M.: The POSTGRES Data Model. In. VLDB '87, 1987.