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:

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA © 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9249-5/22/06...$15.00
https://doi.org/10.1145/3514221.3520176

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,
Function `pack(o1,10)` returns the nested array `[[0,0,3,0],[1,2,0,4]]` of line item numbers, indicating that the third item (part `p3` 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 (of line item numbers, indicating that the third item (part `p3` and `q3`) and to determine the overall volume of a subset of items (`q3`). Note that `q3` 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 database engine:

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

2. When embedded query `SQL(qi,[o1,...,on])` is encountered, the database engine plans query `qi` (or retrieves a formerly constructed plan from a cache), instantiates the plan with parameters `o1, ..., on`, 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 `qi`—the latter is particularly significant if queries are evaluated iteratively (recall `q3` 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 [15].

2. **SNAKES ♠ ON A PLAN**

We demonstrate ByePy, a compiler that translates a Python function, `f`, say, into a single plain SQL-1999 query `Q_f`. Query `Q_f` embodies the entire computation originally performed by the Python code in `f`. If we invoke `Q_f` (instead of `f`) on the Python side, we cross the boundary into the database kernel once: a single plan for `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 `q_pack` assumes the former role of Python function `pack` (follow the path `外交部`, 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 “hoist”, 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 `Qi` 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 `q_pack` (this plan is more complex than those for `Q_i` 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.

2. **SNAKES ♠ ON A PLAN**

We demonstrate ByePy, a compiler that translates a Python function, `f`, say, into a single plain SQL-1999 query `Q_f`. Query `Q_f` embodies the entire computation originally performed by the Python code in `f`. If we invoke `Q_f` (instead of `f`) on the Python side, we cross the boundary into the database kernel once: a single plan for `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 `q_pack` assumes the former role of Python function `pack` (follow the path `外交部`, 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 “hoist”, 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 `Qi` 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 `q_pack` (this plan is more complex than those for `Q_i` 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.
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]:

\[
\text{WITH RECURSIVE } T(...) \text{ AS } \begin{cases}
\{ \text{start} \} \text{ UNION ALL } Q_{\text{rec}}(T) \\
\end{cases}
\]

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 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 (\( \text{if } c_1 \text{ if } c_2 \text{ else } c_3 \)),
- variable assignment (\( v = e, v += e, v[e_1] = e_2 \)) and reference,
- lists (\( [e_1, ..., e_n] \)), indexed access and slicing (\( e_1[e_2], e_1[e_2:e_3] \)),
- stateful list methods (\( \text{pop}, \text{append}, \text{extend} \)),
- a large range of built-in operators and functions (+, \%, **, &, ~, <<, <<=, ==, and, \( \text{len, max, ceil, sqrt, coalesce, ...} \)), and
- embedded read-only queries (\( \text{SQL}(q, [e_1, ..., e_n]) \)).

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 (\( : \text{type} \)), 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.

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

**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 \( f \) 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 \( Q_{\text{start}} \). Once \( Q_{\text{rec}} \) 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 @to_compile
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
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 $\texttt{GOTO}$ or mutual tail recursion, respectively.
• We prepare a formatted display of the SQL code $Q_f$ emitted by ByePy. Seeing $f$ and $Q_f$ 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 $Q_f$ can indeed assume the role of $f$. To facilitate experimentation with the generated code, we wrap $Q_f$ 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 $0.26 \text{ms}$ per invocation if $\texttt{GOTO}$ is inlined into an enclosing SQL query since we bypass any Python-induced latency—see the straight path $\texttt{j}$ 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). These functions will be editable. Entirely new code can be constructed on a blank page at the audience’s request.

REFERENCES


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