Over the last decade, Python emerged as the language of choice for data processing and model building. Yet, the developer productivity of Python comes at the cost of slow execution speed and high demand for resources. Compiling Python to efficient, optimized machine code could help, but writing a general-purpose Python compiler is inherently difficult. Python's dynamic typing, dynamic dispatch, and object format all impede standard compiler optimizations, and efforts to write compilers over two decades have failed to produce substantial speedups for practical data science workloads.

This dissertation introduces two specialized compilers that leverage the high-level structure of a query to compile Python to efficient code for parallel data science workloads. The first exploits the domain-specific program structure of data science pipelines and generates code specialized to a sample of the input data, which allows making assumptions about types and common-case behavior. This results in a novel data-driven compilation approach that produces highly efficient machine code. Our prototype data analytics system, Tuplex, demonstrates that data-driven compilation beats state-of-the-art systems by a factor of 5× to 91× on realistic workloads.

We then explore the idea of hyperspecialization, which generates custom code for disjoint data partitions to avoid sampling errors and improve overall efficiency for heterogeneous datasets. In a distributed setting using serverless functions to avoid sampling errors and improve overall efficiency. Building on Tuplex, we show that this next-generation system, Viton, benefits from hyperspecialization for realistic workloads and reduces cost and runtime by a factor of 2× to 3×. Viton uses serverless Lambda functions to achieve massive parallelism at low latencies together with a more advanced data-driven compilation approach backed by hyperspecialization.

Abstract of "Efficient Data Analytics Using Speculative Compilation Techniques" by Leonhard F. Spiegelberg, Ph.D., Brown University, May 2023.

Efficient Data Analytics Using Speculative Compilation Techniques

by

Leonhard F. Spiegelberg

Master of Science w. Hon., Technische Universität München, Germany, 2016

S. M., Harvard University, Cambridge (MA), 2015

Bachelor of Science, Technische Universität München, Germany, 2013

A dissertation submitted in partial fulfillment of the

requirements for the Degree of Doctor of Philosophy

in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2023

This dissertation by Leonhard F. Spiegelberg is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Malte Schwarzkopf, Director

Recommended to the Graduate Council

Date _____ _____

Tim Kraska (co-advisor), Reader

Date _____ _____

Shriram Krishnamurthi, Reader

Approved by the Graduate Council

Date _____ _____

Thomas A. Lewis
Dean of the Graduate School

Acknowledgements

Above all, I want to thank my advisor Malte Schwarzkopf for his incredible advice, supervision and insights when it comes to conducting systems research at the highest level. Beyond doubt, this dissertation would not stand today without his continuous support and optimism. Malte is a gem of an advisor, and everything a PhD student, who would want to be successful, can hope for in an advisor. I also want to thank Tim Kraska for recruiting me to Brown and for his continuous support as my co-advisor throughout my PhD despite the challenges of remote supervision and two universities being involved. I also owe Shriram Krishnamurthi many thanks for his advice on the work presented in this dissertation, mentorship and trust in me when designing and running a course about Practical Systems Skills from scratch at Brown.

My deepest gratitude goes to Facebook/Meta for supporting my last two years through the Meta fellowship and the exchange with other fellows and Meta researchers.

Throughout the years, I had the privilege to collaborate on the development of two systems with some amazing people, and I am extremely grateful to: Rahul Yesantharao, Ben Givertz, Yunzhi Shao, Colby Anderson, Khosrow Arian, Andy Ly, Raghu Nimmagadda, William Riley, Andrew Wei and Rhea Goyal for their time and contributions to the project.

I also want to thank the great minds and hard working people from my internships at Oracle Labs, Microsoft Research and Snowflake. In particular, I would like to extend my sincere thanks to: Mario Wolzcko, Eric Sedlar, Erkang (Eric) Zhu, Silu Huang, Srinath Shankar for their support, advice and help. I am especially indebted to Srinath Shankar for his long-term vision and support in the final years of my PhD.

My gratitude extends to my incredible HTA Sam Oliphant and the TAs Anina Hitt, Raymond Chao, Hersh Gupta and James Rolfe who made CS6 Practical Systems Skills a most beloved course at Brown.

I also want to express my gratitude to Konstantinos Stylianou for giving me a break on systems research by

learning about and working on a different topic in between, the successful collaboration and sharing an office over a summer at Brown.

Moreover, I want to acknowledge my labmates: Erfan Zamanian, John Meehan, Yeounoh Chung, Philipp Eichmann, Connor Luckett, Dylan Ebert, Justus Adam, Kinan Dak Al Bab, Amir Ilkhechi, Franco Solleza, Zeyuan Shang, Kayhan Dursun, Andrew Crotty, Alex Galakatos and both the ETOS and database group for their thoughts, being around and making the PhD a less lonely journey.

Many thanks also to the staff at Brown for helping me with all sorts of issues throughout my PhD. I want to especially acknowledge here Lauren Clarke for her support in all sorts of matters involving the graduate school and Paul Vars, whose linux sticks helped me more than once out to get a critical experiment done.

Last, I want to especially thank all of my friends from Technische Universität München, Harvard and Brown for their support - without you, life would not as colourful and this endeavour would not have been possible. The ivory tower is real, and your perspective is invaluable to realize what actually does truly matter in life.

Finally, I want to thank the most important people in my life – my family, who have been always there for me through the good and hard times in grad school and without their tireless support and love this PhD would never have happened.

L.F. Spiegelberg

Contents

★ Parts of this dissertation appeared in proceedings of conferences.

List of Tables

List of Figures

1 Introduction

Python won. It has become *the* lingua franca for data science in the 21st century. In the last four annual data science pools conducted by KDnuggets[1], a popular data science community, Python was each time voted as the dominant data science language – favoured and beloved by data scientists around the globe [136, 135, 151]. The immense popularity of Python can be attributed to its beginner-friendliness, high-level abstractions, versatility and vast universe of packages available that are backed by a supportive community [15]. These features make Python attractive, and often the first-choice for prototyping as well as the primary tool to work with large datasets [11]. Indeed, developer time, at which Python excels due to its low startup time (being an interpreted language) and high expressiveness, is sometimes considered to be even more important than raw execution speed [8]. However, the productivity of Python comes at a steep price: language features like dynamic typing, automatic memory management and a single-threaded design using a global interpreter lock result in slow execution speed [60, 205] and high memory consumption, making Python one on the most energy-inefficient languages of the planet [99, 10]. With the projected growth of data centers [71] finding novel solutions to improve Python's efficiency to avoid both frustration when waiting for a query to complete and improving overall efficiency for big data workloads is of paramount interest.

	Pypy [148]	Numba [82]	Pyston[108]	**Tuplex**	GraalPython [123]	Cinder [105]	**Viton**
Year of inception	2007	2012	2014	**2017**	2018	2019[2]	**2021**

Table 1.1: Existing Python compilers and the two novel systems, *Tuplex* and *Viton*, presented in this dissertation.

Due to Python's widespread adoption and significance within the industry, many major tech companies decided to back and develop their own fork of Python in order to speed up their workloads and maintain compatibility for existing code bases (cf. Table 1.1). In this dissertation, we focus on a scenario where users want to process medium to large-scale datasets using logic written in Python. Contrary to its first objective, processing large-scale data with Python was never intended as primary use case, as the original design of Python very much aligned with to be a fast interpreted language with minimal startup time. For a long time – till Python 3.11 – the consensus within the Python community was that micro-optimizations and code specializations should be avoided, as they provide only in minor speed improvements [175, 9, 176]. However, the pressure of other Python implementations utilizing

[1] there was no poll for 2020 due to the pandemic.
[2] open-sourced 2021.

techniques like Quickening [18] implemented successfully in both Pyston [108] and Cinder [13] lead to a shift within the community [168]. With the faster CPython project to achieve a 5× speed improvement of the current state, techniques like quickening and just-in-time compilation are planned to be subsequently rolled out into the reference implementation, CPython, itself [167]. Early results with formerly deemed negligible optimizations showed speed improvements of 10 – 60% [168] which anecdotally are also verified in chapter 2. But even for simple programs rewriting logic in the form of a Python C-extension easily achieves speedups of one to two orders of magnitude over comparable Python code. To maintain fast interpretation of source code, optimizations have to be foregone in favor of fast just-in-time compilation [167]. This makes it unlikely that the faster CPython project will reach comparable performance to a C/C++ compiler that can spend more on compilation and optimizations in the future.

Alternative implementations like the ones in table 1.1 have been successful in speeding up simple, numeric-heavy Python code, but existing alternate drop-in replacements do not work at all or only provide mediocre speed improvements for typical enterprise workloads that are dominated by string operations [17] (cf. chapter 2 for an evaluation).

This results in a classical two-language problem [14]: On the one hand data scientists prefer to write prototypes and pipelines in a high-level language like Python, only to discover at some point performance limitations, that require rewriting most or even all of a pipeline in a more efficient way [93]. Popular choices include using abstractions like SQL or a rewrite using large-scale processing frameworks in more efficient languages like C++, Java, Scala, Go or Rust. Indeed, the availability of a wide range of execution backends prompted the introduction of abstractions like Dataframes that can then be lowered for execution[68, 134].

This process makes it incredibly difficuly to scale pipelines developed as prototypes on a smaller scale to a full-scale production setting [183]. Indeed, the time to product is a partial factor that leads to failure of data science projects to deliver value in commercial settings [72].

1.1 | Background and Challenges

Data-parallel systems offer high-level interfaces to process data. A popular abstraction are LINQ-style MapReduce programs which fueled a big data hype in the early 2010s [44]. [103, 25]. Frameworks built to provide similar APIs allow users to submit custom written user defined functions (UDFs) that are then assembled using higher-level functions like `map`, `filter`, `join` or `aggregate`. Yet, due to the constraints of the Python interpreter most frameworks are architected in a way that they use dedicated Python worker processes to execute arbitrary Python code.

A driver program contains the high-level query, written using the high-level abstractions a framework provides together with the user-defined Python code to be executed by each higher-level operation. After carrying out optional planning and optimizations, a query plan is then executed through the execution engine which in turn invokes multiple

Python processes to evaluate any custom code that is part of the query. This architecture results in several challenges:

Shipping custom Python Code: The code corresponding to the Python UDFs has to be communicated to the worker processes in order to be evaluated. The worker process can either run locally on the same machine where the driver program is invoked, or as a remote process on another machine. This can be done either by forking the original Python process or by serializing the original functions. A popular choice for serialization is Cloudpickle [137] which is build upon the pickle serialization format for arbitrary Python objects. Nearly all modern data processing frameworks use Cloudpickle as it allows to ship functions across a network to remote worker processes. But Cloudpickle does not support all language features, e.g. nested functions or global variables, as upon invocation the environment of an object must be closed. In addition, libraries and dependencies must be manually resolved which is challenging and is often done by loading complete libraries instead of the parts required to execute the serialized function.

Exchanging Python objects: Python objects created as result of the invocation of a UDF as well as conversion of data to input to be passed to Python UDFs result in an overhead. Even though it is possible to write an execution engine using Python objects only, this would be very slow. Instead, most execution engines have a dedicated, faster in-memory format to perform certain higher-level operations (e.g., joins or aggregates) more efficiently. This means that invocation of Python UDFs results in serialization overheads due to object conversion.

Communicating data between workers: Besides conversion of objects, due to the absence of first-class multi-threading support within the Python interpreter, frameworks use an execution engine that spawns multiple processes to parallelize work. As a result, worker processes are forced to communicate with the execution engine and with each other using some form of inter-process communication (IPC). Communication via sockets, e.g. in Spark [204] or shared-memory, e.g in Ray [112], is inefficient as it requires copying the data via syscalls.

Execution of Python Code: Executing the actual UDFs does often require additional logic in the worker process. The natural idea of replacing the interpreter with a more efficient runtime does not necessarily improve overall efficiency as demonstrated in our paper [172]. Most frameworks therefore still use by default the CPython interpreter for its worker processes. While leveraging transpilers like Cython or Nuitka that translate Python to C-code can help to improve runtime performance, the compile time to lower C code to machine code can be larger than the total query time. For example in our paper [172] we observed compilation times of 5s and execution engines using C/C++ code generation report compilation times of around 10s [132]. This is especially problematic when using high degrees of parallelism, as

the additional compile time needed may eclipse the actual query execution time [78]. Therefore, balancing compile and query execution time with the degree of parallelism available is critical to achieve lower end-to-end time for the user.

1.1.1 | *Research problems*

When comparing the performance of a hand-written C++ program to a pipeline executed by a parallel data framework there is a wide performance gap in existing solutions. For example, in chapter 2 we provide an example where a single-threaded C++ program outperforms a tuned Spark program using 16 cores at once by a factor of 1.6×, even though the problem is embarrasingly parallel. To improve performance, the key obstacles to overcome are improving overall UDF execution performance and provide a redesigned system architecture to avoid existing architectural limitations when it comes to parallelism and communication.

With this dissertation, we therefore seek to answer the following questions:

(1) How can we compile Python UDFs to avoid interpreter overheads?

(2) How can we overcome optimization barriers induced by the use of a mix of high-level and UDF-driven operators in an execution engine?

(3) How can we reduce or avoid communication overhead between an execution engine and Python worker processes?

1.2 | Thesis statement

With this work, we introduce *data driven compilation* as a technique to generate more efficient machine from Python code by building a specialized compiler for big data workloads. Our technique is based upon four crucial key observations of properties that UDFs satisfy when they are embedded within a high-level operator graph in a data processing context:

(1) The workflow graph dictates upfront what and how much data to process. This allows to draw statistical conclusions ahead of time using sampling that help determine common-case types and patterns in the data that the compiler can specialize towards. With classical query planning techniques borrowed from database management sytems (DBMSs) this also allows to identify which UDFs lie on a critical path compared to techniques in tracing JITs that first have to detect which and what code regions are hot.

(2) The code of all UDFs is known upfront, i.e. no additional dynamic code is being generated during query execution[3].

[3]We exclude esoteric use cases here, like a UDF using the builtin `eval` function.

(3) UDFs have specific structure: Each UDF takes a *row object* as input and produces another *row object* as output. A high-level operator specifies how to process the collection of *row objects*, e.g. in the case of `filter` whether to keep rows or not.

(4) UDFs are embedded within the high-level graph, are closed upon passing them to an operator and are pure/stateless in the sense that they can be meaningfully reexecuted[4].

These observations can be leveraged to build a special-purpose, integrated UDF/query compiler and lead to the first thesis statement:

> **Statement 1:** It is possible to build a specializing compiler for data processing pipelines that maintains the semantics of Python for a given program and input dataset but significantly improves efficiency by specializing to the input data using an upfront sample of the input data.

However, specialization naturally leads to cases where data does not satisfy constraints that are necessary for optimizations. By utilizing the property that UDFs can be reprocessed and are stateless, a UDF may be always reprocessed in a code path that has fewer restrictions. This observation motivates *dual-case processing*. A novel, coarse-grained execution strategy for which we generate a *normal* and *general* code path. At the beginning of the execution-phase, we classify each input row to belong either to the *normal* or *general* case and process them accordingly. This means we can use a single check per row in comparison to continuous checks (and guards) within tracing compilers. Rows that do not belong to either path process on a *fallback* path using the original CPython interpreter motivating the next thesis statement:

> **Statement 2:** By splitting data upfront into a normal, general and fallback path, expensive deoptimizations can be avoided and Python semantics maintained using *dual-case processing*.

Finally, whereas *dual-case processing* enables to process data more efficiently, sampling all input data may quickly become prohibitive. Especially in a distributed context, waiting for a sampling stage to complete leads to underutilization of massively parallel resources. Therefore, we propose a different strategy: first, use a lightweight sampling approach to get a first result on the overall query structure and then during execution *hyperspecialize* on the input data if it exhibits heterogeneity in order to compensate for suboptimal specialization stemming from the initial sample. This concludes the thesis with:

[4]This means that functions like `random()` are considered *pure/stateless* too.

Statement 3: Adapting code generation to individual data subsets can be used to benefit a distributed execution environment by *hyperspecializing* on-demand on the input data to avoid sampling error and capture specialiazation opportunities within differing marginal data distributions.

1.3 | Thesis contributions and Outline

As core contribution of this thesis, we implemented two novel big data processing frameworks: *Tuplex* and *Viton*, and used them to validate the statements in §1.2. The content of this dissertation is comprised of work that is either currently under submission or has been published before.

- Leonhard Spiegelberg, Tim Kraska, and Malte Schwarzkopf. "Viton: A hyper-specializing execution engine for the serverless age". In: *under submission*. 2023
- Leonhard Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. "Tuplex: Data Science in Python at Native Code Speed". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. Virtual Event, China: Association for Computing Machinery, 2021, 1718–1731. ISBN: 9781450383431. DOI: 10.1145/3448016.3457244. URL: https://doi.org/10.1145/3448016.3457244
- Leonhard F Spiegelberg and Tim Kraska. "Tuplex: robust, efficient analytics when Python rules". In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 1958–1961 (**demo paper**)

The work on *Tuplex* and *Viton* described here also draws from a Bachelor's and a Master's thesis:

- Benjamin Givertz. "Incremental Exception Resolution in Tuplex". Brown University, 2022
- Yunzhi Shao. "Speculative Compilation of Complex UDFs in Python Data Science". Brown University, 2022

I also co-authored the following publications, which informed the work presented in this dissertation, but did not directly contribute to its contents:

- Silu Huang, Erkang (Eric) Zhu, Surajit Chaudhuri, and Leonhard Spiegelberg. *T-ReX: Optimizing Pattern Search on Time Series (Extended Version)*. Tech. rep. MSR-TR-2023-12. Microsoft, 2023. URL: https://www.microsoft.com/en-us/research/publication/t-rex-optimizing-pattern-search-on-time-series-extended-version/ (to appear in SIGMOD'23)
- Kinan Dak Albab, Ishan Sharma, Justus Adam, Benjamin Kilimnik, Aaron Jeyaraj, Raj Paul, Artem Agvanian, Leonhard Spiegelberg, and Malte Schwarzkopf. "Pelton: Privacy-Compliant Storage For Web Applications By Construction". In: *to appear in OSDI'23*. USENIX. 2023
- Konstantinos Stylianou, Leonhard Spiegelberg, Maurice Herlihy, and Nic Carter. "Cryptocurrency competition and market concentration in the presence of network effects". In: *Ledger* 6 (2021), pp. 81–101
- Lorenzo De Stefani, Leonhard F Spiegelberg, Eli Upfal, and Tim Kraska. "VizCertify: A framework for secure visual data exploration". In: *2019 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE. 2019, pp. 241–251

The outline of this dissertation is as follows: Chapter 2 gives backgeound on understanding the architectural choices of existing systems and their resulting limitations that result in subpar performance. Chapter 4 introduces *data-driven* compilation and the design choices that led to the development of *Tuplex*. Chapter 5 verifies statements 1 and 2 from §1.2 through experiments and evaluation. These two chapters are mainly based on work previously published in SIGMOD'21 [172] and VLDB'19 [173]. Chapter 6 introduces *Viton* as a distributed system overcoming the challenges of heterogeneous datasets and addresses statement 3 from §1.2. A detailed evaluation for *Viton* is described in detail in chapter 7. Finally, this dissertation concludes in a discussion of achieved contributions and ideas for future research in chapter 8.

2 Background

Python is an immensely popular language for rapid prototyping and working with large-scale datasets. In fact, a majority of data scientists today prefer to write code in Python, as the language is easy to learn and convenient to use [66]. But the features that make Python convenient for programming—dynamic typing, automatic memory management, and a huge module ecosystem—come at the cost of low performance compared to hand-optimized code and an often frustrating debugging experience. Python code executes in a bytecode interpreter, which interprets instructions, tracks object types, manages memory, and handles exceptions.

Imagine, for example, a data scientist who would like to extract the number of bedrooms from a string like `'3 bds , 1 ba , 1,560 sqft'`. A very straightforward way would be to write a simple UDF in Python like `lambda x: int(x.split('bd')[0].strip())` which transforms the string into an integer. In order to apply the same function to a collection of individual strings, or more generally rows, frameworks like Pandas or Spark are popular choices to provide high-level abstractions that facilitate parallel operations over large datasets.

While this example is extremely simple and amount of lines rather small, there are already multiple challenges to overcome in order to process the dataset efficiently.

Parsing data efficiently and schema inference: Formats like CSV do not specify types upfront. Hence, a system has to detect how many columns are there, which CSV dialect is used, whether the amount of cells per row are always the same, whether a header row is present and if data within a column is homogeneous. I.e., is there a case where in the same column one row stores for example a number and one row a string? While it is possible to treat all columns in a CSV file as string types by default, this is not very user-friendly as most users would need to either explicitly type, validate or parse data manually. This process is not only error-prone but also cumbersome. I.e., in the case of PySpark, the pipeline would only work if there was no escaped newline character `'\n '` (or `'\r '`) present in the file for the custom logic written in fig. 2.1. Spark also does provide a higher-level DataFrame API together with Java code generation, PySparkSQL, but in order to use this API with user-defined functions, users need to convert either to Python objects, or type UDFs explicitly.

Eager vs. lazy execution model: Pandas provides an eager API, that materializes data fully upon each invocation of an operation. In the example above this means that the CSV file is first loaded into memory, and then a new column `'bds'` is created by applying the UDF with the `'facts'` column as input. Yet, frameworks like PySpark

```python
import pandas as pd
df = pd.read_csv('raw_data.csv')
df['bds'] = df['facts'].apply(lambda x: int(x.split('bd')[0].strip()))
```

(a) Pandas

```python
import csv, io
from pyspark.context import SparkContext
sc = SparkContext()

header = sc.textFile('raw_data.csv').map(lambda t: t.split(',')).take(1)[0]

def extract_csv(line):
reader = csv.reader(io.StringIO(line))
cells = next(reader)
return dict(zip(header, cells))

def extract_bds(row):
return int(row['facts'].split('bd')[0].strip())

sc.textFile('raw_data.csv') \
.zipWithIndex().filter(lambda t: t[1] != 0) \
.map(lambda t: t[0]).map(extract\_csv).map(extract_bds).collect()
```

(b) PySpark

```python
from pyspark.sql import SparkSession
from pyspark.sql.types import StringType, IntegerType
from pyspark.sql.functions import udf, col

spark = SparkSession.builder.getOrCreate()

@udf(returnType=IntegerType())
def extract_bds(facts):
return int(facts.split('bd')[0].strip())

spark.read.options(header=True, mode='PERMISSIVE', escape='"') \
   .csv('raw_data.csv').withColumn('bds', extract_bds(col('facts'))).collect()
```

(c) PySparkSQL

```python
import tuplex
ctx = tuplex.Context()
ctx.csv('raw_data.csv') \
   .withColumn('bds',  lambda row: int(row['facts'].split('bd')[0].strip())) \
   .collect()
```

(d) Tuplex

Figure 2.1: Four different ways to process raw data in CSV format to extract information about the number of bedrooms stored in strings. Pandas uses an eager evaluation model, which requires materialization of the full data upon reading. Both Tuplex and PySpark/PySparkSQL build a lazy evaluation graph, which gets evaluated via a `collect` action. PySpark uses tasks to process batches of Python objects, and PySparkSQL uses code generation for the query graph in which Python UDFs may be embedded.

use a lazy execution model as it allows to build an operator graph lazily. The advantage of an operator graph is the ability to optimize on it when the graph is evaluated, usually through special operations called *actions* that force materialization like `collect` or `to_file`. Looking holistic-ally at multiple operations together allows to minimize intermediate materialization points between operators, reorder operators, or skip parts of the input data which are not needed. Note though that using a lazy execution model means that data is read at the state of when an *action* is performed, not when a read command is issued. This means that a framework build using a lazy execution model does have different semantics when it comes to data consistency. Any optimization applied to the operator graph has to ensure that Python semantics are kept as if optimizations were not performed, which can be challenging.

UDFs are optimization barriers: Typically in data processing frameworks, UDFs are treated as black boxes and a framework has no knowledge about the internal workings of a UDF. Within a query graph this effectively means that a UDF becomes an optimization barrier, as a framework can not reason about which columns e.g. a UDF accesses, how long it will take, or whether it could be vectorized. This lack of knowledge makes it difficult to apply many logical operations like projections, reordering or splitting the UDF operator into multiple operators. By treating UDFs as black-boxes a query optimizer may generate a suboptimal plan.

Efficient processing relies on 3rd party libraries: Interestingly though, the same pipeline could be also written in standard Python without any 3rd party library involved (listing 1). While certainly abstractions of a 3rd party library are welcome by many developers to save development time, the major reason to rely on 3rd party libraries for data processing pipelines written in Python is the performance of Python itself. Python is an interpreted language that is executed through a Python runtime. The most widely used Python runtime is the language reference implementation CPython, which till today is the only runtime considered to provide support for all language features. Except for certain scenarios, CPython is also the default implementation deployed in production around the world. But CPython is known to be notoriously slow and unable to take advantage of modern, multi-core architectures.

```
1  import csv
2
3  rows = []
4  with open('raw_data.csv') as f:
5      reader = csv.DictReader(f)
6      for row in reader:
7          row['bds'] = int(row['facts'].split('bd')[0].strip())
8          rows.append(row)
```

Listing 1: The same pipeline as in fig. 2.1 written using the Python standard library.

2.1 | Quick development, slow performance

Many popular frameworks provide a frontend layer in Python which call more efficient primitives written typically directly in C/C++ [27] as CPython allows to call native code loaded dynamically via it's C-API [41]. Popular libraries like Pandas [101], Numpy [53], TensorFlow [1] or PyTorch [130] are implemented with such a C backend. But, using efficient 3rd party libraries limits users to APIs available and makes it difficult and often impossible to write custom code including control flow or in the form of user-defined functions (UDFs).

The main issue is, that the benefit of writing code quickly from a user perspective in Python comes at the cost of performance. This problem appears to be deep-seated: Python has been around for more than 20 years, but its performance over the last decade increased only marginally. This is despite huge efforts of the community to improve performance and tune the reference implementation – i.e., the release version of Python even uses a custom written test-suite to perform profile-guided optimization to tune the executable accordingly. Over the years different alternative implementations like Jython [73] or Pypy [148] have been proposed, but haven't become mainstream and CPython still remain the most widely-used Python language implementation. In this section, we give some background on why replacing CPython is difficult and motivate a different approach of providing a solution for a more specialized, domain-specific scenario of processing large-scale data with Python which is described in detail in chapter 4.

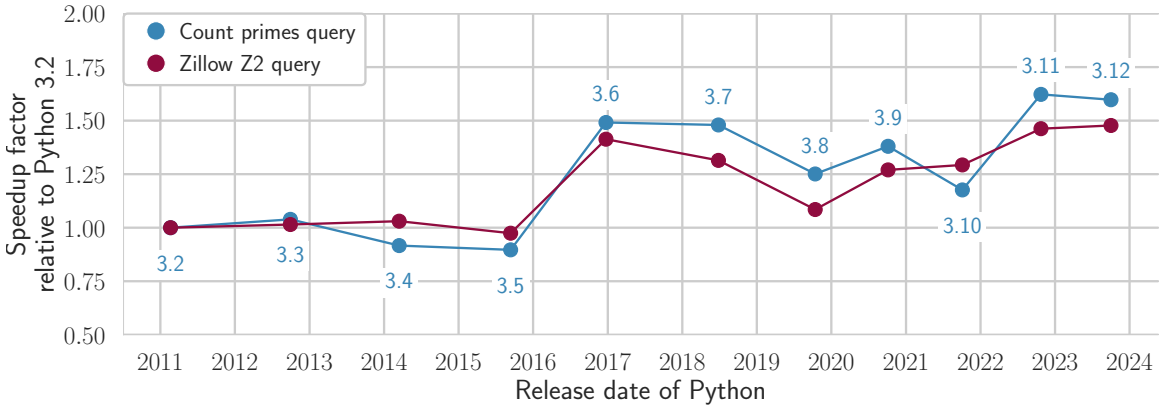


Figure 2.2: More than a decade of development only lead to performance improvements of 48% (Zillow) to 62% (Count primes) of the reference implementation (CPython) compared to its minor release of version 3.2 in 2011.

To demonstrate the challenges of finding a more efficient Python runtime, we discuss anecdotal experiments using two queries. The first query (Zillow) stems from [172] (Z2 in the paper), and executes a mix of filter and map operations over a CSV file to extract information about real estate listings in the Boston area. The second query comes from a blog post about our system Tuplex, which computes prime numbers [115]. Unless otherwise noted we run experiments for the Zillow query with a scaled 1GB input file of the original data on r5d.xlarge EC2 instance

```
    typedef struct _object {
    struct _object *_ob_next;     // part of _PyObject_HEAD_EXTRA macro
    struct _object *_ob_prev;     // part of _PyObject_HEAD_EXTRA macro
    Py_ssize_t ob_refcnt;         // reference count of object
    struct _typeobject *ob_type;  // type object which contains information about
                                   // properties, methods, attributes
} PyObjecdt;
```

Figure 2.3: Each Python object is implemented in CPython using a C struct [98]. `_PyObject_HEAD_EXTRA` form a doubly-linked list of all active heap objects.

and use the input data from the blog post [115] scaled to 16 elements. In fig. 2.2 we show the two queries run with different Python3 minor releases. Each release thereby uses the latest available patch version as of April 9th, 2023[1]. When we look at the result (averages of 10 runs), we can see that there is clear trend of improved performance over time with occasional performance degradation between minor releases for our anecdotal queries. Indeed, jumps in performance can be mostly explained by major initiatives within the Python community: From Python 3.5 to 3.6 performance was improved through a better dictionary implementation [141]. In Python 3.7 attribute accesses and a new fast method call are introduced [140] which has been further refined through a new vectorcall protocol in 3.8 [64]. Python 3.9 brings a new parser [84], and Python 3.10 various micro-optimizations related to type lookups [155]. Yet, the release of 3.11 changed CPython more fundamentally which can be attributed to the goal of improving the reference implementation by a factor of 4–5× as part of the "faster CPython initiative" [167]. The major change here is the introduction of a new, specializing adaptive interpreter to overcome the dynamism of Python, which is one of the major challenges any Python implementation has to face. However, the goal of a 4–5× faster implementation has not been reached and 2 years of implementation resulted in 10–60% performance improvement [156]. This is makes it likely, that reaching the target speedup factor will be a multi-year effort if it will happen ever.

2.2 | Reasons why CPython is slow

There are multiple internal factors that play a role why the reference implementation is slow. Major factors we identified are dynamic typing and dispatch, the in-memory object format the interpreter uses, the interpreter's memory management, and the limitations of using a single global mutex. Especially overcoming dynamic typing and dispatch and keeping compatibility with CPython formats are challenges that make it difficult to build an alternate runtime in the first place to eventually outperform the standard CPython implementation.

[1]Performance of 3.12 was approximated using the alpha release 3.12.0a7 as Python 3.12 is scheduled to be released later in 2023

Dynamic typing and dispatch: Everything in Python is an object associated with and id, type and value [181]. The immutable type of an object thereby defines the behavior at runtime. The behavior of an object may either be predefined in the language runtime[2], by overloading object methods and attributes or by providing an implementation in the form of a C-extension that replaces slots in the Python Object C-structure representation with a custom implementation. A common pattern is to add expensive type checks that are executed during runtime for every single operation to ensure compatibility between two objects. In the case of incompatible objects, exception objects are thrown that may get resolved in outer code blocks. Therefore, it is challenging to determine upfront which objects of which type are produced, as the behavior of objects depends on their runtime type. Implementations can barely make any assumptions and have to keep respecting the semantics of the language reference implementation (CPython).

Python object format: CPython has a standard in-memory object format (cf. fig. 2.3) that needs to be maintained in order to interface with third-party libraries, and that in addition is also required to guarantee semantics for certain operations over objects. For simple objects, the in-memory representation can become quite costly, e.g. a dictionary may require several kilobytes of storage for just a few values.

Memory management: Python uses automatic reference counting together with a garbage collector. Thus, at any time costly garbage collection runs may occur leading, to overall slow-downs. Memory in the interpreter is protected using a single, global mutex, called the GIL (global interpreter lock). This means, that whenever a memory region in the interpreter has to be modified the GIL must be obtained.

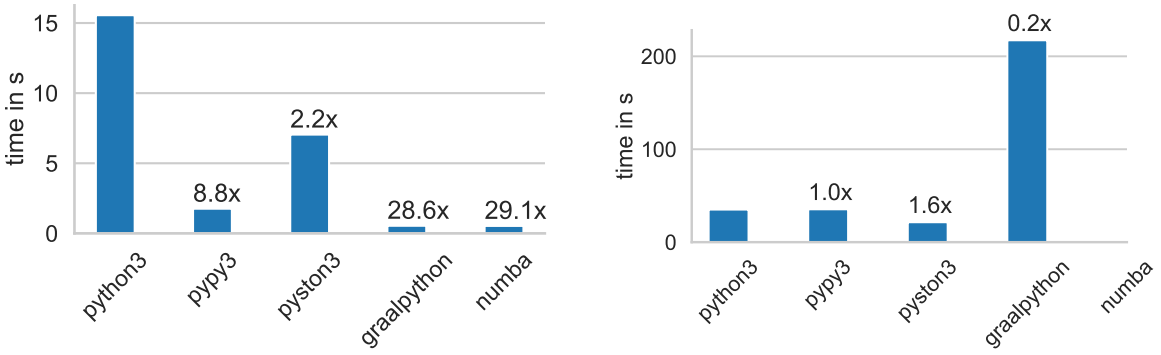
Concurrency: Object access is equally like memory management protected by the GIL. This forces Python execution to be essentially single-threaded. While there's the option to use multi-threading, only IO operations or functions using the C-interface can utilize multiple cores effectively when giving up the lock over the interpreter because only one thread may execute Python code at any time [182].

2.3 | Alternative Python implementations

Nearly every year a new "Python" is announced to solve the performance issues of CPython. Of the various attempts, PyPy is widely regarded as the most mature reimplementation (with more than a decade of continuous innovation and development) of the reference implementation, though it only supports a restricted subset of Python called RPython [148], which encompasses and supports most language features. While there are some differences

[2] the CPython interpreter is usually used as reference implementation to define the default behavior of built-in types like `int`, `float`, `str`, ...

to CPython, PyPy is regarded as a viable option to run many Python programs and promises to deliver improved performance through its just-in-time (JIT) compiler design. Of the more recent projects both Pyston [108], originally supported by Dropbox, and Cinder [**cinder**], sponsored by Meta, are directly build on top of CPython and aim to improve performance through careful optimization and introduction of fast just-in-time compilation for hotspots in the reference implementation. GraalPython is an experimental, but rapidly growing, reimplementation of Python on top of GraalVM and the Truffle framework [124, 123].



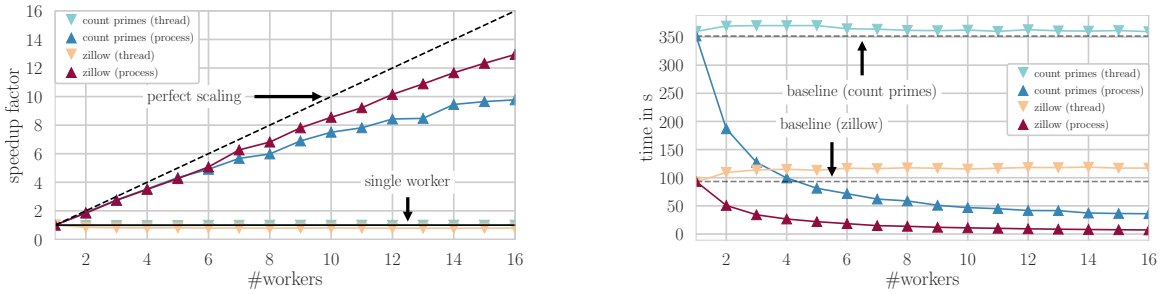
(a) Simple numeric-heavy workload

(b) Big data workload involving strings

Figure 2.4: General-purpose Python alternatives work for simple, numeric functions (2.4a) but fail to speed up typical big data workloads involving string operations (2.4b). Speedup factors relative to the CPython baseline are denoted using × in the graphs. Numba does not support the zillow workload.

In Figure 2.4, we show the performance of alternative Python runtimes anecdotally by showing the speedups they provide on a realistic big data workload cleaning real estate listings [172] using a 1GB input file in CSV format and an artificial, numeric-heavy workload [115] counting prime numbers. For the query counting prime numbers all implementations are able to detect the loop and execute more efficient code for it. Both GraalPython and Numba are also able to leverage vectorization to reach a speedup factor close to 30×. But for workloads involving strings the picture is different: despite different techniques [174, 16] and varying implementations, only a single alternate runtime, Pyston, outperforms the reference runtime CPython for the Zillow query. Furthermore, when scaling the query to a 10GB input file GraalPython exhausted the memory of our machine (256GB) completely and resulted in a crash. While anecdotal, this experiment shows that building a general-purpose Python compiler that can be used as a replacement for CPython is non-trivial and performance improvements are not always guaranteed. While we did not perform a thorough study of the vast universe of alternative python versions, existing literature hints that PyPy can in many scenarios outperform CPython reliably but sacrifices startup cost compared to the reference implementation [146].

This leaves us with the realization, that perhaps a more practical approach is required to scale Python processing for data pipelines.



(a) Speedup **(b)** time

Figure 2.5: Scaling anecdotal queries using Python's standard `multiprocessing` and `concurrent.futures` module compared to a baseline of a single-threaded program in a single process. Using multi-threading leads to contention on the GIL preventing any speedup. Multiprocessing allows to improve performance by distributing the work but suffers both from communication overhead and performance of each process.

2.4 | Multi-core architectures and parallelism

With multi-core architectures being the norm, many data processing problems can be scaled to multi-core architectures by increasing parallelism of the execution engine. Python itself comes with standard modules which allow to parallelize workloads using a thread pool or multiple processes. In fig. 2.5, we show the effect of using Python's builtin modules on overall, end-to-end performance. Due to the embarrassingly-parallel nature of both sample queries, multi-processing achieves better performance using multiple cores compared to a single core. However, adding more and more cores results in diminishing returns at some point as Amdahl's law takes over. In contrary to multi-processing, multi-threading actually results in a worse multi-core performance than when using a single-core. The reason for this is the contention on the GIL and the absence of code sections like I/O interrupts, requests or long-running C-code Python extensions that do not require the GIL. In fact, this leaves multi-processing as the only option for scaling compute across multiple cores in Python.

Higher-level, data-parallel frameworks like Pandas, Spark, Dask or Ray [102, 204, 149, 112] provide an initial abstraction over the data to process, often in the form of dataframes or a parallel collection of objects or tasks that facilitate distributing a workload amongst multiple processes. Besides more convenient abstractions, they also overcome some issues with Python's builtin multiprocessing module, where e.g. functions can be only at the top-level and need to be able to be pickled. A critical piece of any multi-core abstraction is the atomic unit to use in order to distribute work. Spark uses rows as its logical atomic unit, but processes data in batches. Ray uses tasks as primary abstractions, and leaves it to the user to logically subdivide work. Dask, similarly to Ray, is based on tasks, but provides abstractions like bags or dataframes to provide users with automatic data partitioning. Due to the limitations of using threads within Python, whenever data-parallel frameworks want to execute Python code, their only option

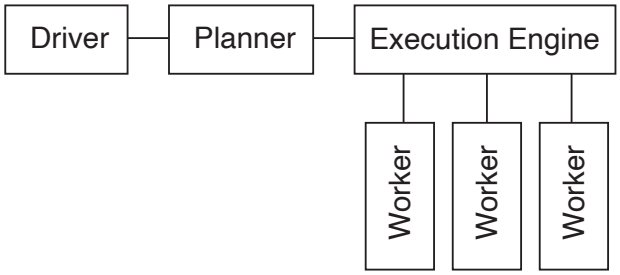


Figure 2.6: Current architecture of frameworks utilizing Python UDFs.

is to rely on multi-processing. As a result, the general architecture of existing frameworks consists usually of a driver that receives the Python program, a planner that translates API calls into a graph and and execution engine that spawns dedicated Python worker processes (fig. 2.6). But using worker processes requires inter-process communication of both the code to execute and the data on which to compute. Spark and Dask use sockets to communicate with worker processes, whereas Ray uses an object storage process which communicates via shared memory via the workers [111]. All frameworks use cloudpickle[137] to communicate Python code, whereas a variety of different in-memory data formats to exchange data are used. These can be either framework specific or open-source formats like Arrow to enable zero-copy integration with other frameworks [6].

However, all of these frameworks require multiple logical units to scale. For single, large objects like matrices found in Machine Learning program this does not work. A simple solution to speedup processing over such large, single objects has been introduced in the form of split annotations, which allow to break up large objects into smaller ones so they fit into the cache [127]. In this dissertation, we focus on how existing, data-parallel frameworks can be improved as they provide *a priori* a suitable abstraction for a user to break up large-scale task into individual subtasks (compared to the *a posteriori* approach of split annotations). Our core abstraction is thereby in the form of rows and columns, similar to a relational database. However, unlike a relational database, we do not require rows to adhere to one static schema. Rather, any row can contain objects of arbitrary types.

The issue with data-parallel frameworks is that while they enable to scale workloads across multiple machines which often is sufficient, they execute Python code rather inefficiently. Nearly every modern DBMS, on the other hand, uses code-generation to provide competitive performance. But code-generation for Python is challenging.

2.5 | Code generation

Consider Spark, which originally was designed to provide MapReduce APIs to run user-provided functions in Scala or Java. These UDFs are executed by first compiling them to JVM bytecode and executing them both via

the Janino Java Compiler (Scala can be compiled to Java via the scala compiler). But support for Python was added to Spark more as an afterthought without the expectation that it would once dominate API usage [200]. Whenever Spark executes a UDF written in Python, it has to materialize a batch of rows to Python objects from its internal format, pass it to a Python worker process, and finally materialize the result of the UDF back to its internal format. Besides the overheads of performing unnecessary data copies there is also a more fundamental architectural concern with this approach when it comes to code generation. The major benefit of code generation is to be able to carry out high-level query optimizations first, before emitting efficient code with a minimal amount of required materialization points [119]. But, introducing a UDF that is executed in a separate worker process introduces an *optimization barrier* into the optimizer. I.e., the query optimizer is not able anymore to perform logical optimizations beyond the UDF boundary, which can result in a suboptimal query plan in the worst case.

In the database community, SQL is the defacto query language which provides convenient high-level abstractions that can be optimized through a query planner to generate an efficient query plan. For this reason projects like ByePy [36, 32] or Froid [144, 143] compile UDFs to SQL as this enables UDFs to be inlined into a SQL query that subsequently can be optimized. However, compiling arbitrary UDFs to SQL is not trivial and mostly works only for UDFs that can be statically typed with a proper object-relational mapping. Moreover, while SQL optimizers have been tuned for years to generate plans for complex operations like JOINs just recently introduction of classical compiler optimizations into SQL optimizers has been proposed [147]. Many SQL optimizers are challenged by complex expressions and do not optimize them well [195].

The promise of achieving superior performance through code generation prompted us to develop a new way of approaching the problem of providing an efficient way to execute Python code at scale. Instead of trying to build another general-purpose compiler that will run into the same obstacles as others before, we decided to focus on a more narrow, constrained problem: building a specialized *compiler* and *execution* engine that leverages the implicit properties of UDFs embedded within a high-level query to make compilation feasible.

3 Approaching Typing in Python

In order to generate efficient code, a compiler needs to know the shape and type of each variable at compilation time. But the primary challenge when it comes to types in Python is that the language itself has no formal specification of a type system nor its operational semantics [138]. Instead, the semantics of Python are mostly defined through the reference implementation [202]. As a consequence, over the years multiple type systems have been proposed to capture the implicit semantics of Python and reason about which results or errors the execution of a Python program should yield. Examples of such retro-fitted type systems are Static Python [97] or popular static type checking tools like MyPy [90], PyType [47] or PySonar2 [194] which each define their own, tool-specific type system and semantics.

Static analyzers and type annotations: Type-checking as part of static analysis can help to ensure that a program adheres to typing rules and does not achieve an undesired state. To perform type checking, checkers need to rely often on users to augment or explicitly annotate identifiers with types as it is difficult to reason about Python programs. For this reason, various tools exist that help to generate type stubs to either be used directly by static analyzers or refined by users to define an initial set of type annotations for a Python program.

However, famously 'Python will remain a dynamically typed language, and the authors have no desire to ever make type hints mandatory, even by convention ' [152]. Type hints are neither required nor enforced during runtime, which means that compilers can not rely on type annotations to be truthful or present. Consider for example listing 2 which is a well-defined Python program. But, up until runtime it is unclear whether the program will produce a `TypeError`, execute safely or which types for `b` would make the program sound as both the slot for addition on the object `a` as well as `b` may get arbitrarily overloaded at any time.

Having a type system in place is desirable as it allows to capture bugs related to unhandled exceptions early on, avoid type errors upfront through static analysis and provide support for linters and code completion within user

```
1  def foo(a: str, b: Any) -> str:
2    # works for 'hello ' + 'world!' but fails for 'hello' + 123
3    return a + b
```

Listing 2: Type annotations were introduced in PEP-484 [152], but are not enforced at compilation time contrary to other languages.

IDEs. Most importantly though, when developing a compiler a type system can help to ensure correctness in terms of translation to machine code and safe execution.

The lack of clearly defined semantics for Python make it challenging to reason about variable types, control flow, exceptions and errors [43]. As a consequence each implementation of Python has two core challenges to solve: 1. Ensure semantics of CPython are adequately captured and/or 2. Provide a reasonable model of operational semantics to follow. This is difficult, as any deviation from the behavior from the CPython reference implementation would violate essentially the language semantics, but nuances of the reference implementation would need to be captured in a type system to guarantee correctness and interoperability with packages.

Consider for example the `is` operator within Python: For integers in the range $-5 <= x <= 256$ calling `is` on integers behaves like the operator == because small-integer objects are cached[1]. Yet, the language reference defines `is` as a test to determine if two objects are the same[2]. For `x=1000` and `y=1000` e.g. evaluates to `False` as two `long` objects get allocated. So it is unclear, what semantics to follow in this case, i.e. whether to model semantics for `is` to reflect checking for object identity in memory or to reflect the quirks of the CPython reference implementation. Not reflecting the quirks could have serious consequences and would mean in the worst case crashes or simply non-interoperability with any Python program written for the CPython implementation originally.

Dynamic language features: Python is a dynamic language that provides extensive reflection and meta programming capabilities [192]. Popular dynamic language features in Python are for example introspection or modification of objects via builtin functions like `getattr`, `hasattr` or `setattr`, as well as code generation (`eval`, `compile`) and dynamic loading of libraries [142]. Because both the behavior and shape of Python objects can be modified at any time, this makes typing control flow and ensuring semantics challenging. Furthermore, the rapidly evolving set of Python features makes it hard for static type analyzers to keep up and guarantee semantics [202].

Restricted language subsets: A common approach when designing Python compilers is to restrict compilation to a subset of the language [148] or merely use Python as a syntax and change semantics to fit a compiler-specific type system. [142] carried out a detailed study to detect how many dynamic features typically are present within both programs and within libraries. Their findings suggest that dynamic features are indeed widely used and restricting users to a subset of the language that can be statically typed was overly conservative. While they confirm that a majority of dynamic features are used during initialization (or module import), they report 16 – 29% of programs still do use dynamic features during runtime.

[1] https://github.com/python/cpython/blob/3.9/Objects/longobject.c#L45
[2] https://docs.python.org/3/reference/expressions.html#is-not

A similar study confirmed the result and also showed that other language features like lambda functions, comprehensions, context managers or generators are widely used beyond initialization [202].

Take-away: Defining sound types for Python statically is hard and may be even impossible. There is no widely accepted, unique type system available for Python that captures all nuances of the reference implementation yet.

3.1 | Sample-based type-inference

Rather than solving the type problem for Python generally, for Tuplex we solve a more domain specific problem of assigning types to variables at query compile time. The key insight is that operating in a large-scale data-processing and compiling only UDFs embedded within high-level operators like `map`, `filter`, or `aggregate` (which each do require a UDF, in the case of aggregate two UDFs for distributing the work) actually is a much more feasible problem than typing an arbitrary Python program. In Tuplex, a high-level operator transforms a sequence of rows into another sequence of rows[3].

Tuplex's UDF compiler is not a general-purpose compiler, but rather a special-purpose compiler for UDFs embedded within high-level operators.

Compared to static Python compilers that fail when encountering Python code that can not be typed, in Tuplex we always execute code and compile only when the UDF compiler can guarantee CPython semantics. Developers can use their UDFs as-is and are not forced to provide explicit type annotations upfront, or write code around the limitations of a compiler which may result in unnecessary code smells and slow development. If compilation is not possible, UDFs are executed in the interpreter.

Indeed, when we consider a typical data pipeline with UDFs, there are a couple properties we leverage to compile UDFs as part of a query.

1. Large number of rows. Each data pipeline has one or more data-source operators. Furthermore, data is processed using a unit of abstraction, commonly referred to as a row that typically has a repeating schema throughout the dataset. Assuming that there is a large number of rows to process, this allows using a sample of rows to estimate a most likely schema that rows adhere to. Data source operators can be either files which provide a static schema, e.g. Arrow, Parquet, ORC, Avro, HDF5, or text-based file formats like CSV or JSON where schemas need to be automatically inferred.

2. Type propagation through chained operators. Types propagate throughout the high-level operator graph. I.e., the output type of a map operator is the input type of the next operator. This dramatically lowers the space of

[3] a join or aggregate operator breaks the ordering assumption for performance reasons.

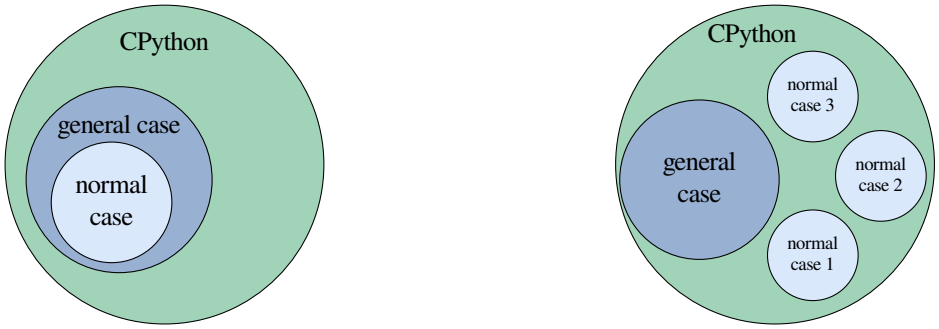
possible type combinations to check for or compile for. It also implies that it is sufficient to perform a type check at the beginning of each data input operator, as the type validity will propagate throughout the graph.

3. Re-execution/Lineage. A high-level assumption of any UDFs submitted is that they can be meaningfully re-executed. This comes from the fact that for fault-tolerance functions passed to high-level operators like `map`/`filter` are assumed to have such behavior so that in the case of failure, the output of a failing operator can be reprocessed and its output considered to be equivalent. In Tuplex, we use this assumption to allow for re-execution of UDFs using different code-paths. I.e., this means that we do not need to de-optimize in-place for each row, but rather can employ a coarse-grained approach by re-processing a row up to three times. With this observation, an optimizer is free to place checks at the beginning of the graph and can avoid having to maintain excessive state for each row line inline caches. I.e., the only state to be maintained is the current row and the state from previous stateful high-level operators like `join`, `aggregate` or `groupby`.

4. Structure of UDFs. When UDFs are used in the context of data pipelines, they follow a pattern of receiving an input row and producing one or more output rows. For simplicity, Tuplex is even more restrictive here and assumes that a UDF always takes one input row and produces one output row. High-level operators then specify how to interpret the result. I.e., a filter checks whether the result is `True` according to CPython rules, or expands a list in the case of `flatMap`. Another important property of UDFs is that they're closed upon being passed to a high-level operator. This means that UDFs do not modify the outer environment, but each UDF defines a local execution environment where the lifetime of any object instantiated, modified or used expires latest when the UDF returns an output row. Temporary objects created by a UDF are not shared between rows, each UDF starts with the same environment when receiving an input row as when the UDF was passed to the high-level operator as parameter.

3.1.1 | *Tuplex's typing approach*

Based on the discussions before, this allows to design an approach to define types in Tuplex: Whenever a query is to be executed an initial sample of the input data is drawn. Then, in a first step the types building a *majority case* form the *general case*, that might be specialized further to the *normal case* to allow further optimization through type specialization. Instead of inferring types through rules or constraints as in classical algorithms like a Hindley-Milner system [178] the type information obtained from an input sample is used and propagated. This is different from approaches like inferring types for Python programs using a SMT solver [54], the Cartesian Product Algorithm used in Starkiller [157] or through explicit type annotations like employed for Numba [83].



(a) Tuplex's code path design **(b)** Viton's code path design

Figure 3.1: Each code path within Tuplex models a different subset of Python. By design, the general-case in Python encompasses the normal-case. In Viton, we relax this assumption to specialize and generate different paths across files. This however means, that deoptimization to from a row that doesn't adhere to the normal-case requires checking before being processed within the general-case.

In Tuplex AST-nodes are in each case automatically annotated with types by passing the output type of the previous operator as input type (in the case of data source operators, a schema is automatically inferred) and applying static typing rules based on the reference CPython implementation. But, not always can a UDF be fully annotated similarly to the limitations a static analyzer faces. For this reason, type annotation in Tuplex is performed in two tiers: First, type annotation is attempted using static type information if feasible. If this fails, a sample of the output of the parent operator is requested and traced through the UDF's AST to determine the paths that the majority case takes. Any paths that do not conform to the majority are removed and guards issued. This is similar to tracing JIT-compilers that first identify hot code regions, collect type information in inline caches and trigger compilation for specific types, thereby adapting to type changes over the time. Contrary to JIT compilers, Tuplex does not keep track of type information during execution (and thus avoids any overheads associated with it). Instead, a sample upfront estimates the most likely code path.

The code paths can be thereby seen as increasingly specialized, i.e. the normal-case is a most specialized code path, and the general-case a most generally typed code-path that still can be compiled. Any rows that do not fall on either code-path are executed through the interpreter. In Tuplex, the normal-case strictly is a specialized version of the general-case. However, the second system in this dissertation, Viton, relaxes this design limitation (cf. fig. 3.1).

3.1.2 | *Types within Tuplex*

Many datasets predominantly use column primitive types like strings, integers or doubles. Motivated by this, Tuplex internally uses monomorphic types with the exception of an optional type `Optional[T]` to allow for null-values that are prevalent across data pipelines. This is also a reasonable choice for the compiled code-paths as [20] [20] report that at least 79% of identifiers in popular Python libraries are type monomorphic. Using preferably monomorphic types allows to eliminate dynamic type checking overheads during runtime that may be a significant

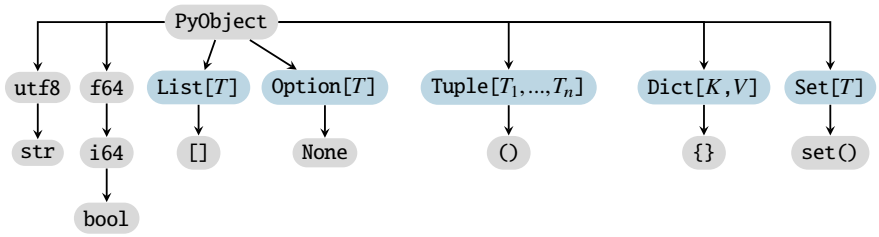


Figure 3.2: Serializable types as defined in and supported within Tuplex's internal type system. Arrows indicate specializations and subtype relations.

type	bool	i64	f64	str	PyObject	None	()	[]	{}
serialized	✓	✓	✓	✓	✓				
variable length				✓	✓				

Table 3.1: Primitive types in Tuplex and their properties. Compound types inherit their properties from the primitive types, e.g. a tuple consisting only of `None` values will not be serialized

factor, e.g. [29] experimentally determined them to be as much as 25% for JavaScript.

For this reason, Tuplex defines for its compiled paths several specialized primitive types like `bool`, `i64`, `f64`, `str` and `None`. There is a unique correspondence between CPython types (defined through which TypeObject is associated with an object) and Tuplex's internal types, like `float` or `None`, but for other types values of objects have to satisfy certain constraints at runtime. For example, consider `float`: Given the CPython reference implementation internally uses 64-bit IEEE floating point numbers (i.e., a `C/C++` **`double`**) to represent objects of type `float` a 1:1 mapping can be established for Python `float` objects. However, for `int` values have to fit into the range of a signed 64bit integer because Python supports arbitrarily large integers for its builtin `int` type. Integers that do not fall into the range representable by 64 bits need to be processed via the interpreter. In addition, Tuplex uses special constant types to identify empty dictionaries, lists, tuples or sets. Using a distinct type for them allows to more easily deduce information and avoid serialization using Tuplex's internal memory format (e.g., a column of all values will not take up any memory when serialized, similarly a column of empty tuples). Besides primitive types, Tuplex also supports compound types like tuples, lists and dictionaries. Viton extends Tuplex's type system by *optimized types* which are further specialized versions of primitive types (e.g., constants for integers) or compound types (e.g., dictionaries with fixed structure). When it comes implicit casts between types, Tuplex strictly follows CPython semantics. For example, technically, and also according to the language, `int` is a subtype of `float` according to Python. However, not all integers can be represented as 64-bit floating point numbers. While this is not an issue for integers requiring 53bit or less when the platform adheres to IEEE-754 standard, this is problematic for e.g. timestamps. An example of a 64bit UTC timestamp was 1682361907670515661 (`'2023-04-24T18:45:07.670515661'`) which can't be represented as a double. As a primary design goal of Tuplex is to avoid type checks at runtime at all costs, we provide a user an option to

specify whether integers should be automatically upcasted to float or not. If set to false, we do not perform any implicit casts in the compiled code-paths and defer execution to the interpreter path reflecting CPython behavior accordingly.

3.1.3 | *Automatic schema inference*

For CSV (and similar) files Tuplex automatically infers a schema using the following heuristic shown in **??**. The

```
def parse(s, null_values={''}):
      # try to parse s as different types
      if s in null_values:
          return None
      try:
          # convert strings like True, t, False, Yes, y, No to bool...
          return to_bool(s.strip())
      except:
          pass
      try:
          return int(s.strip())
      except:
          pass
      try:
          return float(s.strip())
      except:
          pass
      try:
          return json.loads(s.strip())
      except:
          pass
      return s
```

Listing 3: Heuristic used to convert string cells from a CSV file to Python objects.

heuristic first attempts to parse string cells by comparing it to constant strings that form null values (e.g., empty string `''`), then attempts conversion to booleans and finally parses according to the type hierarchy as depicted in fig. 3.2 from most specialized to most general types, reflecting a subtype hierarchy as much as possible. If parsing isn't successful, a string cell is assumed to be of `str` type. Users, however, do have the option to pass explicit type hints to enforce a type. If e.g., a user provides a hint for a column to be of `int` type, Tuplex will attempt to parse the cell contents as integer. If this fails, the above heuristic will be triggered to continue processing. For files or Python objects that do provide types, a mapping to Python objects is established.

3.1.4 | *Detecting the most common type*

The overall goal of finding a schema for each input operator and subsequent pipeline till a pipeline breaker (i.e., for each stage) is to make sure that the majority of rows is processed on the normal case path. Depending on the data, this may or may not be feasible at all. To formalize this, let $\mathcal{S} = \{r_1,...,r_n\}$ be a sample of n rows. For each

row, a type can be detected using a type detection routine t. Some types can be unified (which allows to cover more rows with the same typing schema), but this potentially leads to a performance penalty during execution. A standard example for a type that can be unified is e.g. unifying a null type $t(r_i) = \text{NULL}$ with a primitive type like $t(r_j) = \text{str}$ to an option type $t(r_i) \cup t(r_j) = \text{Option[str]}$. More complex cases involve unifying lists and structured dictionaries. In a sense, there is a trade-off to be made to keep types as specialized as possible while maximizing the probability that a unified type covers as many rows as possible. Using a sample, we can use the Laplace principle to infer the probability $t_k := \mathbb{P}(t(R) = k)$ of a type k to occur within a stage for a random row R.

$$t_k = \mathbb{P}(t(R) = k) \hat{=} \frac{\sum_{i=1,\ldots,n} \mathbb{1}[t(r_i) = k]}{n}$$

More formally, this means that a sampling routine has to solve the following optimization problem for a normal-case threshold δ_{nc} which a user supplies (Tuplex by default uses 0.9). For this let $x_{ij} \in \{0,1\}$ indicate whether type i can be unified with type j and let K be the number of distinct types detected in the sample. This is equivalent to the following linear program:

$$\begin{aligned} \min \quad & \sum_{\substack{i<j \\ i,j=1,\ldots,K}} x_{ij} z_i \\ \text{s.t.} \quad & \sum_{i}^{K} t_i z_i \geq \delta_{\text{nc}} \\ & z_i \in \{0,1\}, \quad i = 1,\ldots,K \end{aligned}$$

which is a classical 0-1 knapsack problem. However, there are still a couple simplifications in the formulation above: 1. each unification is considered to have the same cost to execution. 2. the (significant) execution cost of a row not having a supported schema, which will cause it to land on the fallback path, is neglected. Both issues can be addressed by introducing weights for the cost of each type unification and by letting t_1 w.l.o.g. indicate the probability that a row is processed through the fallback path.

In practice however, solving the program above can become quite costly depending how many types are detected. Empirically, when using, e.g., a dataset containing Github events in JSON, the number of types detected for a small sample can easily be in the thousands. Thus, we use heuristics to compute a likely common path.

3.2 | Types in other data frameworks

Compared to Tuplex, other data processing frameworks often require explicit type annotations when using UDFs.

```python
from pyspark.sql.types import StringType

@udf(returnType=StringType())
def convert_case(s):
  if s is None:
    return None
  res = ''
  arr = s.split(' ')
  for a in arr:
   res += a[0].upper() + a[1:] + ' '
  return res

df = spark.createDataFrame([(1, "john doe")], \
                           ("id", "name"))
df.select(convert_case("name")).show()
```

(a) Spark

```python
import dask.dataframe as dd
import pandas as pd

def convert_case(s):
  if s is None:
    return None
  res = ''
  arr = s.split(' ')
  for a in arr:
    res += a[0].upper() + a[1:] + ' '
  return res

df = pd.DataFrame([(1, 'john doe')], \
                  columns=('id', 'name'))
ddf = dd.from_pandas(df, npartitions=1)

ddf['name'].apply(convert_case, \
                  meta=('name', 'str'))
```

(b) Dask

Figure 3.3: Existing frameworks require type annotations to call Python UDFs.

The reason for this (compared to using multiprocessing, see i.e. §2.4) is that may frameworks use more efficient operators implemented in C (Dask) or even code generation (Spark). But, in order to pass data to these operators it usually has to adhere to a single, supported type. To avoid costly type checks, frameworks rely on explicit type annotations of input data to select the right operator version. When data is encountered that doesn't adhere to a single type, the job usually fails with an exception, making type checks still necessary and resulting in a sub-par user experience. Contrary to this, Tuplex performs a cheap check for each row upfront to determine on which code path to process a row. With this more robust design, Tuplex avoids user frustration and job failure. Exceptions and errors are preserved using Tuplex's execution model.

4 Tuplex

The following chapter and subsequent chapter 5 are based on a previous publication in SIGMOD'21 [172]. We motivate Tuplex by taking a look at how a data scientist would need to implement a PySpark [204] job over flight data [185] that converts a flight's length from kilometers to miles via a UDF after joining the base table containing delay information with a table describing the individual carriers: The code in listing 4 will load data and execute the join using Spark's compiled Scala operators, but must execute the Python UDF passed to the `withColumn` operator in a Python interpreter. This infrastructure imposes a heavy overhead, particularly because here Python *user-defined functions* (UDFs) are inlined in a larger parallel computation graph [204] which requires passing data between the Python interpreter and the JVM [110], and prevents generating end-to-end optimized code across the UDFs. In contrary, an optimized pipeline might apply the UDF to `distance` while loading data from `flights.csv`, which avoids an extra iteration. But the lack of end-to-end code generation prevents this optimization.

Could we instead generate native code (*e.g.*, C++ code or LLVM IR) from the Python UDF and optimize it end-to-end with the rest of the pipeline? Unfortunately, this is not feasible today. Generating, compiling, and optimizing code ahead-of-time that handles all possible code paths through a Python program is not tractable because of the complexity of Python's dynamic typing as described in chapter 2. In the example from listing 4 dynamic typing ("duck typing") requires that code always be prepared to handle *any* type: while the above UDF expects a numeric value for `m`, it may actually receive an integer, a float, a string, a null value, or even a list. The interpreter has to handle these possibilities through extra checks and exception handlers, but the sheer number of cases to deal with makes it difficult to compile optimized code even for this simple UDF. Moreover, it is impossible to statically reason about the types in this example as they depend on the data; a common case in these pipelines.

In addition, dynamic typing forces developers to discover edge cases and bugs by trial and error. Code often fails at runtime—*e.g.*, due to malformed input data, unhandled exceptions, or logic errors—after running for a significant

```
1  carriers = spark.read.load('carriers.csv')
2  fun = udf(lambda m: m * 1.609, DoubleType())
3  spark.read.load('flights.csv')
4     .join(carriers, 'code', 'inner')
5     .withColumn('distance', fun('distance'))
6     .write.csv('output.csv')
```

Listing 4: Example data-science workload applying a simple UDF.

amount of time. Iteratively debugging and handling these exception cases takes valuable developer time.

Speculation simplifies code generation: Tuplex is a new analytics framework that generates optimized end-to-end native code for pipelines with Python UDFs. Its key insight is that targeting the common case simplifies code generation and also provides a convenient abstraction that allows to handle errors efficiently. Developers write Tuplex pipelines using a LINQ-style API [203] similar to PySpark's and use Python UDFs without type annotations. Tuplex compiles these pipelines into efficient native code with a new *dual mode execution model*. Dual-mode execution separates the common case, for which code generation offers the greatest benefit, from exceptional cases, which complicate code generation and inhibit optimization but have minimal performance impact. Separating these cases and leveraging the regular structure of LINQ-style pipelines makes Python UDF compilation tractable, as Tuplex faces a simpler and more constrained problem than a general Python compiler.

4.1 | Dual-mode processing

Making dual-mode processing work required us to overcome several challenges. First, Tuplex must establish what the common case is. Tuplex's key idea is to sample the input, derive the common case from this sample, and infer types and expected cases across the pipeline (cf. chapter 3). Second, the behavior of Tuplex's generated native code must match a semantically-correct Python execution in the interpreter. To guarantee this, Tuplex separates the input data into two row classes: those for which the native code's behavior is identical to Python's, and those for which it isn't and which must be processed in the interpreter. Third, Tuplex's generated code must offer a fast bail-out mechanism if exceptions occur within UDFs (*e.g.*, a division by zero), and resolve these in line with Python semantics. Tuplex achieves this by adding lightweight checks to generated code, and leverages the fact that UDFs are stateless to re-process the offending rows for resolution. Fourth, Tuplex must generate code with high optimization potential but also achieve fast JIT compilation, which it does using tuned LLVM compilation.

Tuplex achieves by generating three code paths: Two compiled code paths for the compiled mode, and one code path for the interpreter. Generating the interpreter path is necessary to keep compatibility with the compiled code paths for exception handling and to interact seamlessly between the different code paths. The data flow of Tuplex can be seen in fig. 4.1.

First, Tuplex takes a sample from the beginning of the first input file to the pipeline, and uses it to infer the common-case types and control flow through Python UDFs. This inference allows Tuplex then to *specialize* the code to the common-case input data by making assumptions about control flow (*e.g.*, which branch of an `if` statement will run) and about data types (cf. figure 4.1). Tuplex then generates LLVM IR for two code paths: (*i*) a *fast path*

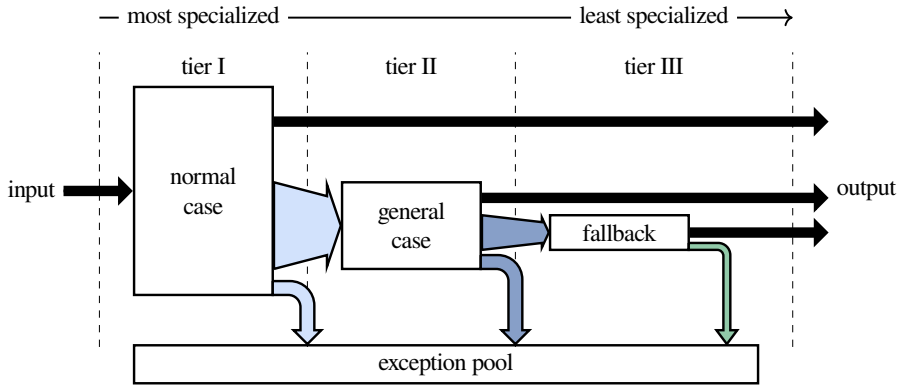


Figure 4.1: Tuplex code path execution model: the normal case code path is the most aggressively optimized path and may require many assumptions in order to compile. The general case code path is the code path that still can get compiled but has the least restrictions. The interpreter path processes any rows as a fallback for the compiled code paths that do not pass the checks for either the normal or general case. Normal and general case form the compiled mode and the interpreter path the fallback mode of Tuplex.

specialized to the common-case assumptions from the sample; and (*ii*) a *slow path* that makes fewer assumptions, but contains more instructions. Tuplex dispatches work to multiple executors to process the input dataset in parallel using these two code paths. If these code paths' assumptions fail to hold for a record, or if it fails processing on both paths, Tuplex falls back to the Python *interpreter*. In addition non-conforming rows are either relegated to the more general code path, the interpreter or in the case of exceptions captured in an exception pool.

Exception handling: Dual mode processing enables compilation, but has another big advantage: it can help developers write more robust pipelines that never fail at runtime due to dirty data or unhandled exceptions. Tuplex detects exception cases, resolves them via slow-path execution if possible, and presents a summary of the unresolved cases to the user. This helps prototype data wrangling pipelines, but also helps make production pipelines more robust to data glitches. The focus of Tuplex is primarily on multi-threaded processing on a single server, but we also implemented a experimental, distributed backend for Tuplex is a distributed system with AWS Lambda functions. Viton builds on top of Tuplex and massively extends the experimental backend by supporting all code paths and through the introduction of hyperspecialization (cf. 6).

Contributions: In summary, in this chapter we make the following principal contributions:

1. We combine ideas from query compilation with speculative compilation techniques in the *dual-mode processing* model for data analytics: an optimized common-case code path processes the bulk of the data, and a slower fallback path handles rare, non-conforming data without inhibiting optimization.
2. We observe that data analytics pipelines with Python UDFs—unlike general Python programs—have sufficient

structure to make compilation without type annotations feasible.

3. We build and evaluate Tuplex, the first data analytics system to embed a Python UDF compiler with a query compiler.

4.2 | Comparing Tuplex to existing solutions

Many prior attempts to speed up data science via compilation or to compile Python to native code exist, but they fall short of the ideal of compiling end-to-end optimized native code from UDFs written in natural Python. We discuss key approaches and systems in the following; Table 4.1 summarizes the key points.

System Class	Examples	Limitations
Tracing JIT Compilers	PyPy [148], Pyston [108]	Require tracing to detect hotspots, cannot reason about high-level program structure, generated code must cover full Python semantics (slow).
Special Purpose JIT Compilers	Numba [82], XLA [88], Glow [153]	Only compile well-formed, statically typed code, enter interpreter otherwise; use their own semantics, which often deviate from Python's.
Python Transpilers	Cython [12], Nuitka [55]	Unrolled interpreter code is slow and uses expensive Python object representation.
Data-parallel IRs	Weld [126], MLIR [87]	No compilation from Python; static typing and lack exception support.
SQL Query Compilers	Flare [33], Hyper [119]	No Python UDF support.
Simple UDF Compiler	Tupleware [21]	Only supports UDFs for which types can be inferred statically, only numerical types, no exception support, no polymorphic types (*e.g.*, `NULL` values).

Table 4.1: Classes of existing systems that compile data analytics pipelines or Python code. All have shortcomings that either prevent full support for Python UDFs, or prevent end-to-end optimization or full native-code performance.

Python compilers. Building compilers for arbitrary Python programs, which lack the static types required for optimizing compilation, is challenging. PyPy [148] reimplements the Python interpreter in a compilable subset of Python, which it JIT-compiles via LLVM (*i.e.*, it creates a self-compiling interpreter). GraalPython [124] uses the Truffle [69] language interpreter to implement a similar approach while generating JVM bytecode for JIT compilation. Numba [82] JIT-compiles Python bytecode for annotated functions on which it can perform type inference; it supports a subset of Python and targets array-structured data from numeric libraries like NumPy [5].

All of these compilers either myopically focus on optimizing hotspots without attention to high-level program structure, or are limited to a small subset of the Python language (*e.g.*, numeric code only, no strings or exceptions). Pyston [108] sought to create a full Python compiler using LLVM, but faced memory management and complexity challenges [107], and offers only a 20% performance gain over the interpreter in practice [109].

Python Transpilers. Other approaches seek to cross-compile Python into other languages for which optimizing compilers exist. Cython [12] unrolls the CPython interpreter and a Python module into C code, which interfaces with standard Python code. Nuitka [55] cross-compiles Python to C++ and also unrolls the interpreter when cross-compilation is not possible. The unrolled code represents a specific execution of the interpreter, which the compiler may optimize, but still runs the interpreter code, which compromises performance and inhibits end-to-end optimization.

Data-parallel IRs. Special-purpose native code in libraries like NumPy can speed up some UDFs [67], but such pre-compiled code precludes end-to-end optimization. Data-parallel intermediate representations (IRs) such as Weld [126] and MLIR [87] seek to address this problem. Weld, for example, allows cross-library optimization and generates code that targets a common runtime and data representation, but requires libraries to be rewritten in Weld IR. Rather than requiring library rewrites, Mozart [128] optimizes cross-function data movement for lightly-annotated library code. All of these lack a general Python UDF frontend, assume static types, and lack support for exceptions and data type mismatches.

Query compilers. Query compilers turn SQL into native code [2, 206, 79, 166, 179], and some integrate with frameworks like Spark [33]. The primary concern of these compilers is to iterate efficiently over pre-organized data [170, 77], and all lack UDF support, or merely provide interfaces to call pre-compiled UDFs written e.g. in C/C++.

Simple UDF compilers. UDF compilation differs from traditional query compilation, as SQL queries are declarative expressions. With UDFs, which contain imperative control flow, standard techniques like vectorization cannot apply. While work on peeking inside imperative UDFs for optimization exists [58], these strategies fail on Python code. Tupleware [21] provides a UDF-aware compiler that can apply some optimizations to black-box UDFs, but its Python integration relies on static type inference via PYLLVM [56], and it lacks support for common features like optional (`None`-valued) inputs, strings, and exceptions in UDFs. Tuplex supports all of these.

Exception handling. Inputs to data analytics pipelines often include "dirty" data that fails to conform to the input schema. This data complicates optimizing compilation because it requires checks to detect anomalies and exception handling logic. Load reject files [106, 23, 139] help remove ill-formed inputs, but they solve only part of the problem, as UDFs might themselves encounter exceptions when processing well-typed inputs (*e.g.*, a division by zero, or `None` values). Graal speculatively optimizes for exceptions [30] via polymorphic inline caches—an idea also used in the V8 JavaScript engine—but the required checks and guards impose around a 30% overhead [28]. Finally, various

dedicated systems track the impact of errors on models [80] or provide techniques to compute queries over dirty data [193, 198], but they do not integrate well with compiled code.

Speculative processing. Programming language research on speculative compilation pioneered native code performance for dynamically-typed languages. Early approaches, like SELF [19], compiled multiple, type-specialized copies of each control flow unit (*e.g.*, procedure) of a program. This requires variable-level speculation on types, and results in a large amount of generated code. State-of-the-art tracing JITs apply a dynamic variant of this speculation and focus on small-scale "hot" parts of the code only (*e.g.*, loops).

A simpler approach than trying to compile general Python is to have Python merely act as a frontend that calls into a more efficient backend. Janus [62, 63] applies this idea to TensorFlow, and Snek [26] takes it one step further by providing a general mechanism to translate imperative Python statements of any framework into calls to a framework's backend. While these frameworks allow for imperative programming, the execution can only be efficient for Python code that maps to the operators offered by the backend. To account for Python's dynamic types, such systems may have to speculate on which backend operators to call. In addition, the backend's APIs may impose in-memory materialization points for temporary data, which reduce performance as they add data copies.

In big data applications, efficient data movement is just as important as generating good code: better data movement can be sufficient to outperform existing JIT compilers [128]. Gerenuk [117] and Skyway [120] therefore focus on improving data movement by specializing serialization code better within the HotSpot JVM.

Tuplex. In Tuplex, UDFs are first-class citizens and are compiled just-in-time when a query executes. Tuplex solves a more *specialized* compilation problem than general Python compilers, as it focuses on UDFs with mostly well-typed, predictable inputs. Tuplex compiles a fast path for the common-case types (determined from the data) and expected control flow, and defers rows not suitable for this fast path to slower processing (*e.g.*, in the interpreter). This simplifies the task sufficiently to make optimizing compilation tractable.

Tuplex supports natural Python code rather than specific libraries (unlike Weld or Numba), and optimizes the full end-to-end pipeline, including UDFs, as a single program. Tuplex generates at most three different code paths to bound the cost of specialization (unlike SELF); and it speculates on a per-row basis, compared to a per-variable basis in SELF and whole-program speculation in Janus. Tuplex uses the fact that UDFs are embedded in a LINQ-style program to provide high-level context for data movement patterns and to make compilation tractable. Finally, Tuplex makes exceptions explicit, and handles them without compromising the performance of compiled code: it collects exception-triggering rows and batches their processing, rather than calling the interpreter from the fast path.

4.3 | Tuplex Overview

Tuplex is a data analytics framework with a similar user experience to *e.g.*, PySpark, Dask, or DryadLINQ [103]. A data scientist writes a processing pipeline using a sequence of high-level, LINQ-style operators such as `map`, `filter`, or `join`, and passes UDFs as parameters to these operators (*e.g.*, a function over a row to `map`). *E.g.*, the PySpark pipeline shown in §1 corresponds to the Tuplex code: Like other systems, Tuplex partitions the input data

```
c = tuplex.Context()
carriers = c.csv('carriers.csv')
c.csv('flights.csv')
 .join(carriers, 'code', 'code')
 .mapColumn('distance', lambda m: m * 1.609)
 .tocsv('output.csv')
```

Listing 5: Tuplex API example to read two CSV files, join them on the carrier code column, apply the miles to kilometers conversion UDFs introduced in chapter 4, and save the output as CSV file.

(here, the CSV files) and processes the partitions in a data-parallel way across multiple executors. Unlike other frameworks, however, Tuplex compiles the pipeline into end-to-end optimized native code before execution starts. To

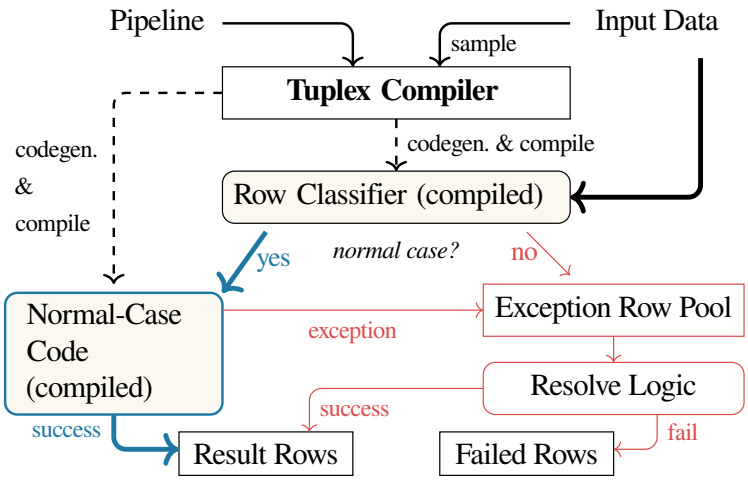


Figure 4.2: Tuplex uses an input sample to compile specialized code for normal-case execution (blue, left), which processes most rows, while the exception-case (red, right) handles the remaining rows. Compiled parts are shaded in yellow.

make this possible, Tuplex relies on a **dual-mode processing** model structured around two distinct execution modes:

1. an optimized, *normal-case* execution; and
2. an *exception-case* execution.

To establish what constitutes the normal case, Tuplex samples the input data and, based on the sample, determines the expected types and control flow of the normal-case execution. Tuplex then uses these assumptions to generate and

optimize code to classify a row into normal or exception cases, and specialized code for the normal-case execution. It lowers both to optimized machine code via LLVM.

Tuplex then executes the pipeline. The generated classifier code performs a single, cheap initial check on each row to determine if it can proceed with normal-case execution. Any rows that fail this check are placed in an exception pool for later processing, while the majority of rows proceed to optimized normal-case execution. If any exceptions occur during normal-case execution, Tuplex moves the offending row to the exception pool and continues with the next row. Finally, after normal-case processing completes, Tuplex attempts to resolve the exception-case rows. Tuplex automatically resolves some exceptions using general, but slower code or using the Python interpreter, while for other exceptions it uses (optional) user-provided resolvers. If resolution succeeds, Tuplex merges the result row with the normal-case results; if resolution fails, it adds the row to a pool of failed rows to report to the user.

In our example UDF, a malformed flight row that has a non-numeric string in the `distance` column will be rejected and moved to the exception pool by the classifier. By contrast, a row with `distance` set to `None`, enters normal-case execution if the sample contained a mix of non-`None` and `None` values. However, the normal-case execution encounters an exception when processing the row and moves it to the exception pool. To tell Tuplex how to resolve this particular exception, the pipeline developer can provide a resolver: Tuplex then merges the resolved

```
# ...
 .join(carriers, 'code', 'code')
 .mapColumn('distance', lambda m: m * 1.609)
 .resolve(TypeError, lambda m: 0.0)
# ...
```

Listing 6: Tuplex allows users to add special high-level operators `resolve` and `ignore` to deal with exception within UDFs. This helps to keep logic concise, but also informs Tuplex's query and UDF compiler to place the logic on the slow compiled code path or interpreter code path and assume that these exceptions are unlikely to occur.

rows into the results. If no resolver is provided, Tuplex reports the failed rows separately. An extension to Tuplex where exceptions are incrementally resolved has been implemented in [46] [46].

4.4 | Design

Tuplex's design is derived from two key insights. First, Tuplex can afford slow processing for exception-case rows with negligible impact on overall performance if such rows are rare, which is the case if the sample is representative. Second, specializing the normal-case execution to common-case assumptions simplifies the generated logic by deferring complexity to the exception case, which makes JIT compilation tractable and allows for aggressive optimization.

4.4.1 | *Abstraction and Assumptions*

Tuplex's UDFs contain natural Python code, and Tuplex must ensure that their execution behaves exactly as it would have in a Python interpreter. We make only two exceptions to this abstraction. First, Tuplex never crashes due to unhandled top-level exceptions, but instead emulates an implicit catch-all exception handler that records unresolved ("failed") rows. Second, Tuplex assumes that UDFs are pure and stateless, meaning that their repeated execution (on the normal and exception paths) has no observable side-effects.

The top-level goal of matching Python semantics influences Tuplex's design and implementation in several important ways, guiding its code generation, execution strategy, and optimizations.

4.4.2 | *Establishing the Normal Case*

The most important guidance for Tuplex to decide what code to generate for normal-case execution comes from the observed structure of a sample of the input data. Tuplex takes a sample of configurable size every time a pipeline executes, and records statistics about structure and data types in the sample, as follows.

Row Structure. Input data may be dirty and contain different column counts and column orders. Tuplex counts the columns in each sample row, builds a histogram and picks the prevalent column structure as the normal case.

Type Deduction. For each sample row, Tuplex deducts each column type based on a histogram of types in the sample. If the input consists of typed Python objects, compiling the histogram is simple. If the input is text (*e.g.*, CSV files), Tuplex applies heuristics. For example, numeric strings that contain periods are floats, integers that are always zero or one and the strings "true" and "false" are booleans, strings containing JSON are dictionaries, and empty values or explicit "NULL" strings are `None` values. If Tuplex cannot deduce a type, it assumes a string. Tuplex then uses the most common type in the histogram as the normal-case type for each column (except for null values, described below).

This *data-driven* type deduction contrasts with classic, static type inference, which seeks to infer types from program code. Tuplex uses data-driven typing because Python UDFs often lack sufficient information for static type inference without ambiguity, and because the actual type in the input data may be different from the developer's assumptions. In our earlier example (§4.3), for instance, the common-case type of `m` may be `int` rather than `float`.

For UDFs with control flow that Tuplex cannot annotate with sample-provided input types, Tuplex uses the AST of the UDF to trace the input sample through the UDF and annotates individual nodes with type information. Then, Tuplex determines the common cases within the UDF and prunes rarely visited branches. For example, Python's power operator (`**`) can yield integer or float results, and Tuplex picks the common case from the sample trace execution.

Option types (`None`). Optional column values (i.e, "nullable") are common in real-world data, but induce potentially expensive logic in the normal case. Null-valued data corresponds to Python's `None` type, and a UDF must be prepared for *any* input variable (or nested data, *e.g.*, in a list-typed row) to potentially be `None`. To avoid having to check for `None` in cases where null values are rare, Tuplex uses the sample to guide specialization of the normal case. If the frequency of null values exceeds a threshold δ, Tuplex assumes that `None` is the normal case; and if the frequency of null values is below $1-\delta$, Tuplex assumes that null values are an exceptional case. For frequencies in $(1-\delta,\delta)$, Tuplex uses a polymorphic optional type and generates code for the necessary checks.

4.4.3 | *Code Generation*

Having established the normal case types and row structure using the sample, Tuplex generates code for compilation. At a high level, this involves parsing the Python UDF code in the pipeline, typing the abstract syntax tree (AST) with the normal-case types, and generating LLVM IR for each UDF. The type annotation step is crucial to making UDF compilation tractable, as it reduces the complexity of the generated code: instead of being prepared to process any type, the generated code can assume a single static type assignment.

In addition, Tuplex relies on properties of the data analytics setting and the LINQ-style pipeline API to simplify code generation compared to general, arbitrary Python programs:

1. UDFs are "closed" at the time the high-level API operator (*e.g.*, `map` or `filter`) is invoked, *i.e.*, they have no side-effects on the interpreter (*e.g.*, changing global variables or redefining opcodes).
2. The lifetime of any object constructed or used when a UDF processes a row expires at the end of the UDF, *i.e.*, there is no state across rows (except as maintained by the framework).
3. The pipeline structures control flow: while UDFs may contain arbitrary control flow, they always return to the calling operator and cannot recurse.

Tuplex's generated code contains a *row classifier*, which processes all rows, and two code paths: the optimized *normal-case* code path, and a *general-case* code path with fewer assumptions and optimizations. The general-case path is part of exception-case execution, and Tuplex uses it to efficiently resolve some exception rows.

Row Classifier. Tuplex must classify all input rows according to whether they fit the normal case. Tuplex generates code for this classification: it checks if each column in a row matches the normal-case structure and types, and directly continues processing the row on the normal-case path if so. If the row does not match, the generated classifier code copies it out to the exception row pool for later processing. This design ensures that normal-case processing is focused on the core UDF logic, rather including exception resolution code that adds complexity and disrupts control flow.

Code Paths. All of Tuplex's generated code must obey the top-level invariant that execution must match Python semantics. Tuplex traverses the Python AST for each UDF and generates matching LLVM IR for the language constructs it encounters. Where types are required, Tuplex instantiates them using the types derived from the sample, but applies different strategies in the normal-case and general-case code. In the normal-case code, Tuplex assumes the common-case types from the sample always hold and emits no logic to check types (except for the option types used with inconclusive null value statistics, which require checks). The normal-case path still includes code to detect cases that trigger exceptions in Python: *e.g.*, it checks for a zero divisor before any division.

By contrast, the general-case path always assumes the most general type possible for each column. For example, it includes option type checks for all columns, as exception rows may contain nulls in any column. In addition, the general-case path embeds code for any user-provided resolvers whose implementation is a compilable UDF. But it cannot handle all exceptions, and must defer rows from the exception pool that it cannot process. The general-case path therefore includes logic that detects these cases, converts the data to Python object format, and invokes the Python interpreter inline.

Tuplex compiles the pipeline of high-level operators (*e.g.*, `map`, `filter`) into stages, similar to Neumann [119], but generates up to three (fast, slow, and interpreter) code paths. Tuplex generates LLVM IR code for each stage's high-level operators, which call the LLVM IR code previously emitted for each UDF. At the end of each stage, Tuplex merges the rows produced by all code paths.

Memory Management. Because UDFs are stateless functions, only their output lives beyond the end of the UDF. Tuplex therefore uses a simple slab allocator to provision memory from a thread-local, pre-allocated region for new variables within the UDF, and frees the entire region after the UDF returns and Tuplex has copied the result.

Exception handling. To simulate a Python interpreter execution, the code Tuplex generates and executes for a row must have no observable effects that deviate from complete execution in a Python interpreter. While individual code paths do not always meet this invariant, their combination does. Tuplex achieves this via *exceptions*, which it may generate in three places: when classifying rows, on the normal-case path, and on the general-case code path. Figure 4.3 shows how exceptions propagate rows between the different code paths.

Rows that fail the row classifier and those that generate exceptions on the normal-case code path accumulate in the exception row pool. When Tuplex processes the exception row pool, it directs each row either to the general-case code path (if the row is suitable for it) or calls out to the Python interpreter. Any rows that cause exceptions on the general-case path also result in a call into the interpreter. An interpreter invocation constitutes Tuplex's third code path, the *fallback code path*. It starts the UDF over, running the entire UDF code over a Python object version of

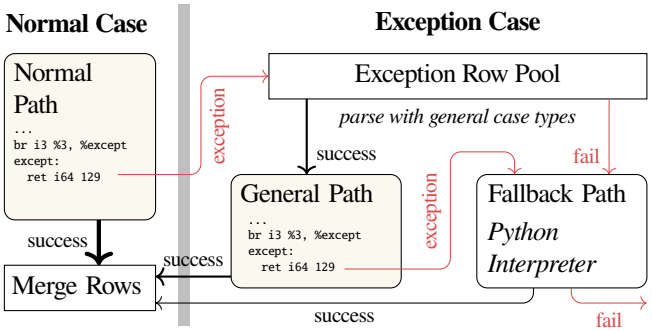


Figure 4.3: Tuplex's exception case consists of a compiled general path and a fallback path that invokes the Python interpreter. Exceptions (red) move rows between code paths.

the row. Finally, if the pipeline developer provided any resolvers, compilable resolvers execute on the general-case code path, and all resolvers execute on the fallback path. If the fallback path still fails, Tuplex marks the row as failed.

Consequently, Tuplex may process a row a maximum of three times: once on the normal-case path, once on the general-case path, and once on the fallback path. In practice, only a small fraction of rows are processed more than once.

4.4.4 | *Execution*

Tuplex executes pipelines similar to a typical data analytics framework, although customized to handle end-to-end UDF compilation. Tuplex has a logical planner, which applies logical optimizations (*e.g.*, operator reordering and filter pushdown); a physical planner, which splits the pipeline execution into distinct stages; and a UDF compiler, which handles the actual code generation. However, the typing requirements of Tuplex's dual-mode processing model permeate all these components. For example, the logical planner also types the UDFs according to the normal-case types deduced from the sample in order to allow for type-aware logical optimizations.

Stages. A *stage* is a sequence of operators, including UDFs, that is bounded on either side by an operator that consumes materialized data from memory or requires generating it. Examples of such operators include inputs, joins, aggregations, and outputs. Stages are also the unit of code generation: Tuplex generates and executes a normal-case and an exception-case code path for each stage. The materialized output of a stage may initially consist only of normal-case result rows, though some operators require immediate production and materialization of resolved exception-case rows too (see §4.4.5).

Tuplex delineates stages similarly to HyPer [119]. Tuplex makes stages as long as possible to facilitate compiler optimizations, and so that rows are processed through many UDFs while in CPU cache. Ideally, most input rows proceed through a single, highly-optimized stage that ends with the pipeline's materialized output.

4.4.5 | *Joins*

Tuplex uses a hash join, which materializes records on one side of the join (the "build" side) and streams rows on the other side to look up into the hash table. Tuplex chooses the smaller side as the build side and terminates a stage at the materialized join input.

This standard design, however, requires adaptation for dual-mode processing. A classic data-parallel join works because the data on both sides of the join is partitioned by the same key. For join $A \bowtie B$ between relations A and B, it suffices to join each $A_i \bowtie B_i$. But in dual-mode execution, each partition of A is itself split into normal-case rows $NC(A_i)$ and exception-case rows $EC(A_i)$, and likewise for B. For correct results, Tuplex must compute each pairwise join:

$$NC(A_i) \bowtie NC(B_i) \cup NC(A_i) \bowtie EC(B_i) \cup$$
$$EC(A_i) \bowtie NC(B_i) \cup EC(A_i) \bowtie EC(B_i)$$

To compute the joins between normal-case and exception-case rows, Tuplex would have to execute all three code paths for *both* join inputs and materialize the input rows in memory. This conflicts with the goal of long stages that keep caches hot on the normal path and avoid unnecessary materialization. Instead, Tuplex executes all code paths for the build side of the join and resolves its exception rows before executing any code path of the other side. If the build side is B and the result of resolving exception rows of B_i is $R(B_i) = NC(B_i) \cup \text{resolve}(EC(B_i))$, Tuplex then executes $NC(A_i) \bowtie R(B_i)$ as part of a longer stage and without materializing $NC(A_i)$.

4.4.6 | *Aggregates*

Dual-mode processing works for aggregations as long as the aggregation function is associative. Tuplex separately aggregates normal-case rows and, subsequently, exception-case rows via the general and fallback code paths; in a final merge step, it combines the partial aggregates into a final result. This merging of partial aggregates happens at the end of the stage after resolving exception rows.

Aggregations are compatible with Tuplex's assumption that UDFs are stateless, as the framework tracks the accumulated state across rows. To make this work, the aggregation operator needs to take a UDF with a row argument and an accumulator argument, and return an updated accumulator. For example, `.aggregate`'s UDF signature is `lambda acc, r: acc + r['col']`, where `acc` is an accumulator (*e.g.*, an integer, a list or a more complicated object like a nested tuple or dictionary). Tuplex's runtime is responsible for managing the memory of `acc`, and the UDF remains stateless.

4.4.7 | *Optimizations*

Tuplex applies several optimizations to the normal-case path.

Logical optimizations. Pushing selective operators (*e.g.*, filters, projections) to the start of the pipeline is a classic database optimization. Yet, systems that treat Python UDFs as black box operators cannot apply this optimization across UDFs. Tuplex's logical planner analyzes UDFs' Python ASTs to determine which input objects are preserved, dropped, and modified by each UDF. Based on this knowledge, Tuplex then reorders operators to preserve columns only as long as needed. Another, more complex optimization pushes UDFs that modify a column past any operators and UDFs that do not read it. This helps *e.g.*, push UDFs that rewrite non-key columns below joins, which is a good choice if the join is selective. Crucially, this optimization is possible because Tuplex analyzes the Python UDFs.

UDF-specific optimizations. Tuplex applies standard compiler optimizations like constant folding to Python UDFs. In addition, Tuplex applies optimizations specific to UDFs as part of a LINQ-style pipeline. For example, Tuplex rewrites dictionaries with string keys known at compile time into tuples (avoiding string operations); Tuplex flattens nested tuples to avoid pointer indirection; and Tuplex optimizes for the common case in nullable values, *i.e.*, column types can get specialized to `NULL`, `Option[T]` or `T`.

Code generation optimizations. On the normal-case path, Tuplex removes any code related to types that it classified as exceptions. Consider, for example, `lambda m: m * 1.609 if m else 0.0`: with an input sample of mostly non-null floats, Tuplex removes code for integer-to-float conversion, null checks, and the `else` branch from the normal-case path. This reduces the generated code from 17 LLVM IR instructions (5 basic blocks) to 9 IR instructions (1 basic block). If the common-case input is null, Tuplex simplifies the normal-case path to 3 IR instructions that return zero.

Compiler optimizations. Once Tuplex has generated LLVM IR for the normal-case path, it applies several LLVM optimizer passes to the code. In particular, we use the Clang 9.0 pass pipeline equivalent to `-O3` which are applied for all UDFs and operators inside a stage.

However, since Tuplex's generated code must match Python semantics, not all compiler optimizations are valid. For example, some optimizations to speed up floating point math (equivalent to the `-ffast-math` C compiler flag) change the handling of `NaN` values in ways that fail to match Python. Tuplex avoids these optimizations.

4.5 | Implementation

We implemented a prototype of Tuplex in about 65k lines of C++. Our prototype uses LLVM 9's ORC-JIT to compile the generated LLVM IR code at runtime. It is implemented as a C-extension (shared library) which users import as a Python module or from a Jupyter Notebook. Tuplex provides a shell in CPython interactive mode and a web UI with a history server, which developers can use to inspect their pipelines' execution and any failed rows generated. Tuplex is available open-source under `https://github.com/tuplex/tuplex`. To ensure competitive performance, we made several design choices.

Multithreaded Execution. On a single server, our prototype runs executors in a thread pool. Executors process input data partitions in individual tasks, which run identical code. Each thread has its own bitmap-managed block manager for memory allocation. When invoking the fallback path, Tuplex acquires the global interpreter lock (GIL) of the parent Python process.

Distributed Execution. Tuplex's techniques apply both on a single server and in a distributed setting, where many servers process parts of the input data in parallel. For datasets that require this scale-out data parallelism, our prototype supports executing individual processing tasks in serverless AWS Lambda functions over data stored in S3. Tuplex divides each stage into many data-parallel tasks and runs each task in a Lambda function, which reads its input from S3 and writes its output back to S3. The driver machine generates LLVM IR, initiates, and supervises the Lambdas. Each Lambda receives the LLVM IR code of its task from the driver, lowers the IR to machine code, and executes the machine code over its input data.

Exception handling. Tuplex implements exception control flow on the normal-case and general-case paths via special return codes. We found that this is 30% faster than the "zero-cost" Itanium ABI exception handling [95], and allows more optimization than `setjmp/longjmp` (SJLJ) intrinsics [96].

Limitations. Our prototype supports compiling optimized code for many, but not all Python language features. The prototype currently supports compiling integer, float, string, and tuple operations, as well as essential dictionary and list operations. UDFs can be passed either as lambda functions or regular functions and may contain optional type annotations. The prototype supports variables, simple list comprehensions, control flow, random number generation, and regular expressions. It does not yet support while loops, generator expression, try-except, sets, async expressions, classes, objects, nested functions and external modules. For unsupported language features, Tuplex falls back on

running the UDF in the Python interpreter. We believe that support for many missing core Python features could be added to our prototype with additional engineering effort.

Our prototype also does not focus on external modules, which could be compiled but often already come with their own native-code backends. Linking Tuplex's generated LLVM IR with the LLVM IR code produced by library-oriented compilers such as Weld [126], Numba [82] or Bohrium [81] should be feasible in future work.

5 Tuplex – Evaluation

We evaluate Tuplex with three representative pipelines and with microbenchmarks of specific design features. Our experiments seek to answer the following questions:

1. What performance does Tuplex achieve for end-to-end data science pipelines, compared to both single-threaded baselines and widely-used parallel data processing frameworks? (§5.1)
2. What is the cost of Tuplex's code paths, and of exception handling? (§5.1.4)
3. How does Tuplex's performance compare to off-the-shelf Python compilers, such as PyPy, Cython, and Nuitka; and to state-of-the-art query compilers, such as Weld [126] and Hyper [76]? (§5.1.5)
4. What factors affect Tuplex's performance, and what is the impact of optimizations enabled by Tuplex's dual-mode processing model? (§5.2)
5. How does Tuplex perform when operating distributedly across many servers? (§5.3)

Setup. In most experiments, Tuplex and other systems run on an `r5d.8xlarge` Amazon EC2 instance (16-core Xeon Platinum 8259CL, 2.50 GHz; hyperthreads disabled) with 256 GB RAM, and 2 NVMe SSDs. The input data is CSV-formatted UTF-8 text. We compare Tuplex against Dask (2021.03) and Spark (PySpark, v2.4.7) on Ubuntu 20.04. All systems use 16-way parallelism. All numbers are averages of at least five runs with warmed-up OS caches.

Our focus is Tuplex's performance on a multi-core server, a common medium-scale analytics setup [33]. But the systems we compare against support scale-out across servers, so we also compare Tuplex's prototype AWS Lambda backend to Spark (§5.3).

5.1 | End-to-End Performance

We measure Tuplex's end-to-end performance using three data science pipelines, and with the datasets shown in Table 5.1.

Zillow. Zillow is a real estate directory website whose listings are uploaded by individual brokers. We scraped 38,570 Boston area listings [165], scaled the data to 10 GB, and cleaned it for performance experiments to avoid failures in Spark and Dask. The two queries extract information like the number of bedrooms, bathrooms, and the price from textual data and filter for all houses (Z1) or condos currently for sale (Z2). Each version involves eleven

Dataset	Size	Rows	Columns	Files
Zillow	10.0 GB	48.7M	10	1
Flights	5.9 GB	14.0M	110	24
	30.4 GB	69.0M	110	120
Logs	75.6 GB	715.0M	1	3797
311	1.2 GB	197.6M	1	1
TPC-H (SF=10)	1.5 GB	59.9M	4	1

Table 5.1: Dataset overview (smaller join tables excluded).

Python UDFs, which perform value conversions, multiple substring extractions, and several simple lookups, as well as filtering out implausible records. The UDF's operators can execute as a single, large pipelined stage.

Flights. We modeled this workload after a Trifacta tutorial [48] and extended it by joining with additional airport and airline data from other sources (743 KB [129] and 82 KB [184]). The pipeline has one inner and two left joins, as well as UDFs to reconstruct values from multiple columns which can't be easily expressed in a SQL query. We ran this pipeline on ten years (2009-2019) of CSV data [185].

Weblogs. Based on a Spark use case [22], this pipeline extracts information from twelve years of Apache web server logs obtained from a U.S. university. It converts the Apache log format into a relational representation, and retains records for potentially malicious requests. We extended the original query by an inner join with a list of bad IPs [116] and anonymize any personally-identifiable URLs by replacing usernames (*e.g.*, "`~alice`") with random strings.

311 and TPC-H. We use the Pandas cookbook [34] data cleaning query for 311 service requests, which yields a set of unique ZIP codes, to compare to Weld [126]. Finally, we also run microbenchmarks with TPC-H Q6 and Q19 to measure Tuplex's performance compared to Hyper [76], a state-of-the-art SQL query compiler.

5.1.1 | *Zillow: String-heavy UDFs.*

In this experiment, we compare Tuplex to other frameworks using the Zillow pipeline. This pipeline contains eleven UDFs, which use Python idioms for substring search (*e.g.*, `"bd" in s`, or `s.find("bd")`), string splitting, normalization (`s.lower()`), and type conversions (`int`, `float`).

We consider two row representations: (*i*) as Python tuples, and (*ii*) as Python dictionaries (hashmaps). The dictionary representation simplifies code by using column names, but typically comes with a performance overhead. Tuplex allows either representation and compiles both representations into identical native code.

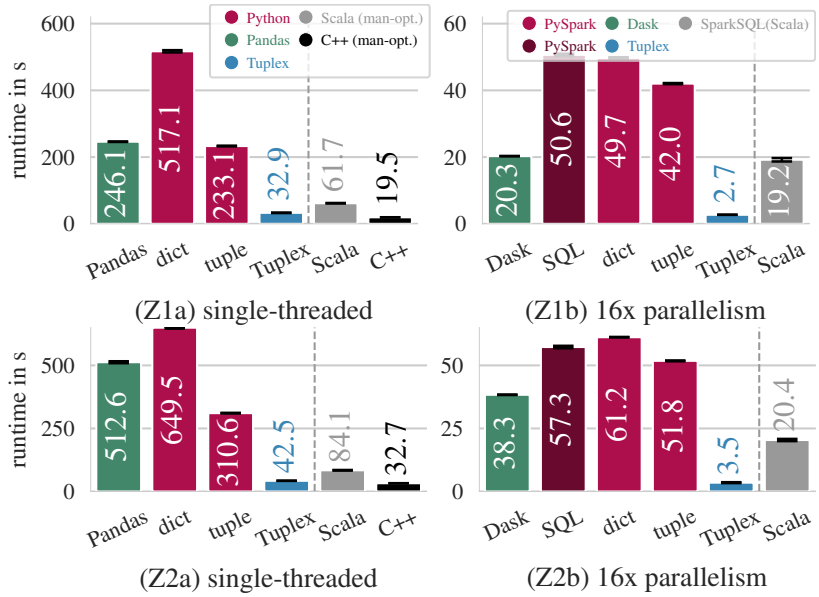


Figure 5.1: Tuplex outperforms single-threaded and parallel alternatives by 5.8×–18.7× when running the Zillow pipeline over 10G of input data, and comes close to hand-tuned C++.

Single-threaded execution. We compare standard CPython (v3.6.9), Pandas (v1.1.5), and hand-optimized C++ (via gcc v10.2) and Scala (v2.12.10) baselines to Tuplex configured with a single executor. Tuplex's end-to-end optimized code might offer an advantage over CPython and Pandas, which call into individual native-code functions (*e.g.*, `libc` string functions) but cannot optimize end-to-end. Tuplex should ideally come close to the hand-optimized C++.

Figure 5.1 shows our results. As expected, the CPython implementation with rows represented as dictionaries is substantially slower (about 2×) than the tuple-based implementation. Pandas, perhaps surprisingly, is about 5.5% slower than tuple-based CPython in Z1, and 65% slower than tuple-based CPython in Z2. While Pandas benefits from a faster CSV parser, an efficient data representation (`numpy` arrays), and specialized native-code operators for numeric computation, its performance suffers because UDFs require converting between `numpy` and Python data representations. Z2 filters fewer rows early than Z1, which exacerbates this UDF-related cost. Finally, Tuplex completes processing in 33–43 seconds, a speedup of 5.8×–18.7× over the CPython and Pandas implementations. This is 1.9× faster than a single-threaded baseline written in pure Scala, and 1.3–1.7× slower than the hand-optimized C++ implementation. However, this overstates Tuplex's overhead: in Tuplex, the compute part of Z1 (*i.e.*, excluding I/O) takes 11.2s, 29% slower than the C++ implementation (8.7s); Z2 sees a 0.5% slowdown (19.8s *vs.* 19.7s).

Data-parallel execution. Next, we benchmark Tuplex against widely-used frameworks for parallel processing of large inputs: PySpark (v2.4.7) and Dask (2021.03). We configure each system for 16-way parallelism: PySpark uses 16

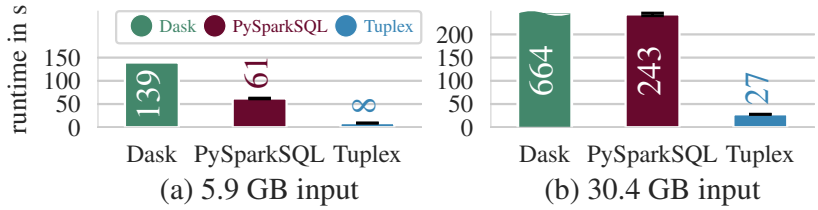


Figure 5.2: Tuplex achieves speedups of 7.6–24.6× over PySparkSQL and Dask on the flights pipeline.

JVM threads and 16 Python processes for UDFs; Dask uses 16 worker processes; and Tuplex uses 16 executor threads. We benchmark PySpark both with RDDs [204] and with the more efficient SparkSQL operators [7]. Neither PySpark nor Dask compile UDFs to native code or optimize across UDFs, which indicates that Tuplex should outperform them.

Figure 5.1 confirms that this is the case: Tuplex outperforms the fastest PySpark setup by 15.5× and Dask by 7.5× in Z1. For Z2, Tuplex is 14.5× faster, as the compiled UDFs process more rows. Compared to the single-threaded execution, Tuplex achieves a speedup of 12.2× when using 16 threads (for Z1). We also ran the pipeline in SparkSQL with Scala UDFs rather than Python UDFs, which keeps computation within the JVM and avoids overheads of calling into Python. Tuplex's end-to-end optimized code is still 5.8–7.1× faster.

These results confirm that Tuplex's code generation and end-to-end optimization offer performance gains for UDF-heavy pipelines. In §5.1.5, we compare Tuplex to other Python compilers, and §5.2 drills down into the factors contributing to Tuplex's performance.

5.1.2 | *Flights: Joins and Null Values.*

We repeat the comparison between Tuplex, Spark, and Dask for the flights pipeline. This pipeline contains three joins, and the dataset has "sparse" columns, *i.e.*, columns that mostly contain null values, while others have occasional null values complemented by extra columns (*e.g.*, diverted flights landing at a different airport). Tuplex infers the normal-case null value status for each column from its sample, and defers the more complicated logic needed to resolve exception rows to the general-case code path. 2.6% of input rows violate the normal-case and get handled by the general-case code path in Tuplex. Spark and Dask handle null values inline in UDF execution, and we use PySparkSQL, which compiles the query plan (though not the UDFs) into JVM bytecode. Figure 5.2 shows the results for two years' worth of data (5.9 GB) and ten years (30.4 GB). PySparkSQL outperforms Dask by 2.3–2.7× because of its compiled query plan and more efficient join operator. Tuplex, despite its unoptimized join operator, still achieves a 7.6–9× speedup over PySparkSQL (17.4–24.6× over Dask) because it compiles and merges the UDFs, and processes the bulk of the data through a single, end-to-end optimized stage (we break this down in §5.2.2).

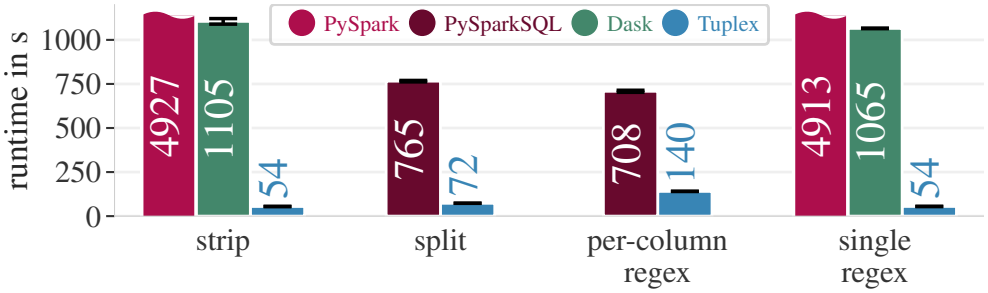


Figure 5.3: Tuplex outperforms Spark and Dask by 5.1–91× on the weblogs pipeline; all Tuplex variants perform similarly. PySparkSQL only supports per-column regexes.

5.1.3 | *Log Processing: Regex and Randomness.*

We use the weblogs pipeline to investigate how Tuplex's compiled code compares to special-purpose operators designed to accelerate common UDF functionality in existing frameworks. The pipeline splits an input log line into columns, and then rewrites one of those columns with a random string if it matches a username pattern (cf. listing 7): We consider three settings for the log line splitting operation:

1. natural Python using string operations (`strip`/`split`);
2. per-column regular expressions; and
3. a single regular expression.

Natural Python requires UDFs in all systems, but we also wrote an equivalent query using SparkSQL's native string functions (*i.e.*, the query executes entirely within the JVM). PySparkSQL also has a native operator for regular

```
1  def randomize_udf(x):
2    r = [random_choice(LETTERS) for t in range(10)]
3    return re_sub('^/~[^/]+', '/~' + ''.join(r), x)
```

Listing 7: Anonymization UDF in the Logs query.

expressions (`regexp_extract`). It only supports per-column regular expressions (second setting), but the operator applies the regular expression in the JVM, rather than in Python. Finally, all systems currently require UDFs when using a *single* regular expression.Tuplex supports all three approaches.

We would expect Python UDFs (both `strip`/`split` and regex-based) in Spark and Dask to be slowest. PySparkSQL's native regex operator and the `split`-like SQL query should outperform them. A good result for Tuplex would show performance improvements in all three setups, as Tuplex end-to-end compiles and optimizes each setting for this pipeline. The input in our experiment is 75.6 GB of logs (715M rows). For Dask, we excluded 31.7M rows (4.5%, 4 GB) of the data because they triggered a known bug in the inner join [187].

Figure 5.3 reports the results organized by setting. The PySpark pipelines with two UDFs are slowest at about 80 minutes, while Dask UDFs are roughly 4× faster (18 min). Dask is more efficient because it executes the entire pipeline in Python, avoiding costly back-and-forth serialization between the JVM and Python workers. However, when PySparkSQL keeps the log line splitting in the JVM—either using string functions (PySparkSQL (split)) or via per-column regexes—runtime reduces to about 12 minutes. This happens because SparkSQL can generate JVM bytecode for most of the pipeline (except the randomization UDF) via its whole-stage code generation [201]. Tuplex, on the other hand, completes the pipeline in one minute both using natural Python and with a regular expression. Per-column regular expressions slow Tuplex down by a factor of two, but it still outperforms PySparkSQL by 5.1×; likewise, Tuplex's `split`-based pipeline is 10.6× faster than PySparkSQL's equivalent native SQL query. This difference comes, in part, because Tuplex compiles both UDFs to native code, while PySpark can only use compiled code for line-splitting. When we subtract the anonymization UDF runtime in both systems, Tuplex is still about 8× faster than PySparkSQL. The remaining speedup comes from Tuplex's end-to-end optimization, and from using PCRE2 regular expressions: in our microbenchmarks, PCRE2 is 8.85× faster than `java.util.regex`, which Spark uses.

Tuplex's fastest pipelines (single regex, strip) outperform the best PySpark and Dask setups by 13× and 19.7×. Tuplex supports logical optimizations unavailable to Dask and Spark that improve performance further, which we discuss in §5.2.1.

5.1.4 | *Exception Handling*

Tuplex speeds up processing of common-case rows by deferring exception-case rows to slower code paths. Exception rows arise either because of malformed ("dirty") input data, or because they don't match the common case in Tuplex's sample (*e.g.*, due to type mismatches). We now measure the cost of exception row handling. We run Z2 on the original, uncleaned Zillow dataset (scaled to 10 GB). 25% of the 56M input rows are exception rows with malformed data. We compare three setups: (*i*) ignoring and discarding all exception rows; (*ii*) the developer manually resolving exceptions in UDFs; (*iii*) using Tuplex's resolvers (§**??**), both with compiled resolvers (on the general path) and resolution in the interpreter (fallback path). Our prototype runs a single-threaded Python interpreter for the fallback path, but this could be parallelized, so we assume a hypothetical, ideal 16× speedup to obtain a lower bound on fallback path runtime. Ignoring exception rows should be the fastest, while a good result for Tuplex would show manual and automatic resolution achieve similar performance, and a low overhead for handling exception rows. Figure 5.4 shows a breakdown of Tuplex's execution time in each case. Ignoring all exceptions is fastest, since it merely skips the rows. Manual resolution adds an 8% overhead, but requires laborious changes and makes the UDFs much more complex. Tuplex's compiled resolvers come within 0.3% of the hand-crafted resolution, with the

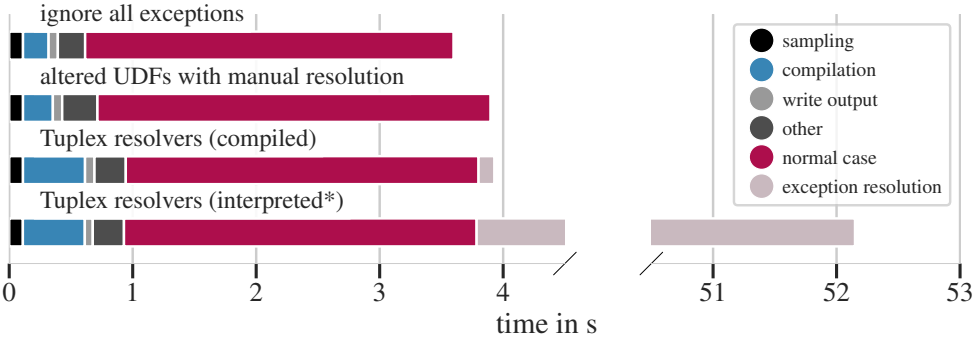


Figure 5.4: Tuplex's exception resolution adds little overhead (0.3%) with compiled resolvers for Z2 on "dirty" data (25% of rows malformed). The interpreter (*) bar is a lower bound that assumes ideal 16× speedup over single-threaded interpreter.

overhead owed to increased LLVM compilation time. When we force all resolution onto the fallback path, however, it adds a 13× overhead, as 25% of rows are now processed in the Python interpreter. This shows that Tuplex's compiled general-case path is crucial for good performance on exception-heavy workloads.

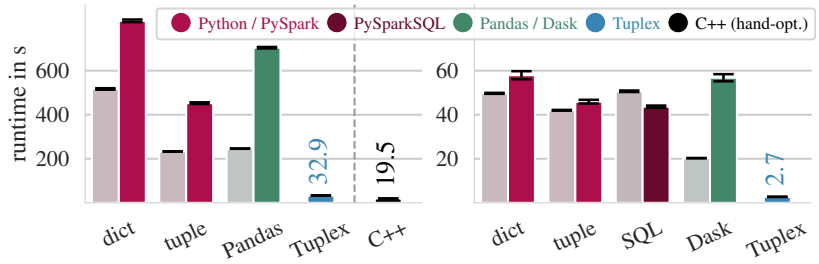
Processing a single row on the normal path takes 0.8µs. The compiled general path takes 0.28µs per row on average, as most exception rows are discarded early. To measure the full cost, we replaced all exception rows with synthetic data that proceeds all the way through the pipeline; in this case, the compiled general path takes 1.3µs (*vs.* 299µs/row in the interpreter).

5.1.5 | *Comparison To Other Systems*

We now compare Tuplex to systems that generate and (JIT-)compile efficient native code for Z1. Z2 yields similar results (omitted). First, we compare Tuplex to general Python compilers, which compile arbitrary Python programs.

PyPy. PyPy [148] is a tracing JIT compiler that can serve as a drop-in replacement for the CPython interpreter. It detects hot code paths (usually loops), JIT-compiles a specialized interpreter and caches the hot paths' native code. We configured Pandas, PySpark and Dask to use PyPy (v7.3.3) instead of CPython to measure how well PyPy performs on UDFs, and run the Zillow pipeline in the same setups as before. Even though PyPy is still bound to Python's object representation and has limited scope for end-to-end optimization, the hope is that JIT-compiling the hot code paths will improve performance.

Figure 5.5 shows that this is actually not the case. PyPy is slower than interpreted Python in all settings, by between 3% and 3.18×; only with PySparkSQL it comes close to interpreted Python. Profiling with cProfile [40] suggests that PyPy has a variable impact on UDF performance: of twelve UDFs, seven are faster (13%–11.6×) with PyPy, and five are 26%–2.8× slower. The one UDF that benefits substantially (11.6×) merely forms a tuple; for



(a) single-threaded (b) 16x parallelism

Figure 5.5: The PyPy3 general-purpose JIT fails to accelerate the Z1 query, and degrades performance by up to 3×. Dark bars use PyPy, light bars use the CPython interpreter (Fig. 5.1).

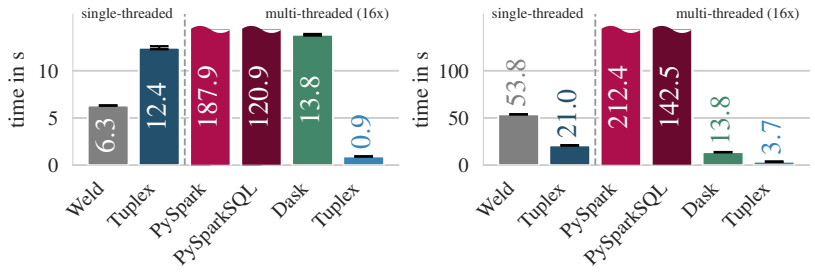
	System	Runtime	Compile time
CPython (interpreter)		233.1 s	–
Python compilers	Cython	195.3 s	6.5 s
	Nuitka	192.5 s	9.4 s
	Tuplex	32.3 s	**0.2** s
Hand-optimized C++		**19.2** s	7.5 s

Table 5.2: Tuplex runs the Z1 query 6× faster than Cython and Nuitka, and compiles 32–47× faster than alternatives.

others, even superficially similar string-processing UDFs exhibit varying performance. We attribute this to PyPy JIT-compiling and caching only some code paths, but not others. The 3× slowdown for Pandas and Dask is due to PyPy3's poor performance when invoking C extension modules [180]. Tuplex is 14–24× faster.

Cython and Nuitka. Nuitka and Cython emit C/C++ files that contain unrolled calls to C functions which power the CPython interpreter. Compiling this file into a shared library object produces a drop-in replacement for a Python module. We used Nuitka (v0.6.13) and Cython (v0.29.22) to transpile the Python module to C for Z1 and compile it with gcc v10.2. This eliminates the cost of Python byte code translation and allows the C compiler to optimize the whole pipeline. We run the resulting module over 10 GB of input data, and compare single-threaded runtime to interpreted CPython and Tuplex.

Table 5.2 shows runtimes and compile times. Nuitka and Cython's compiled code runs 17% faster than interpreted Python, but is still over 6× slower than Tuplex. Tuplex outperforms Nuitka and Cython because it replaces C-API calls with native code, eliminates dispensable checks and uses a more efficient object representation than Cython and Nuitka, which use CPython's representation. Cython and Nuitka also have 32–47× higher compile times than Tuplex. They take about a second to generate code, with the rest of the compile time taken up by the C compiler (gcc v10.2). Tuplex generates LLVM IR, which is faster to compile than higher-level C/C++, and also compiles 37× faster than gcc compiles the C++ baseline.

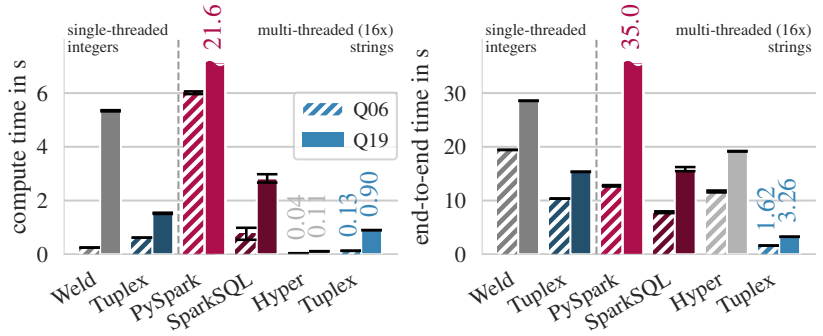


(a) query-time only (b) end-to-end

Figure 5.6: For the 311 data cleaning pipeline, single-threaded Tuplex comes within 2× of Weld and outperforms all parallel systems. Tuplex outperforms Weld by 2× end-to-end because Tuplex inlines the aggregation in its generated parser.

Data-parallel IR: Weld. Weld is a data-parallel IR that admits optimizations like vectorization or loop fusion across libraries [126]. Weld serves as a backend to ported existing libraries such as Pandas [102], while Tuplex is a complete data analytics system, but both execute compiled native code. We compare Tuplex's performance to Weld's on the **311** data cleaning workload [34] and **TPC-H Q6** and **Q19**. Q6 and Q19 perform simple filters and aggregations and are a challenging workload for Tuplex, which shines at string-heavy workloads with row-level UDFs and does not yet support vectorized (SIMD) compilation of UDFs. We compare to Weld v0.4.0; since Weld's multithreaded runtime was removed in v0.3.0 [125], we compare single-threaded performance. In addition, we preprocessed the inputs to contain only the required columns and converted all dates to integers. In the single-threaded case all string-typed columns in Q6 and Q19 were transformed to integers, because Weld lacks automatic projection pushdown and has limited string processing capabilities. Because Weld does not have a native CSV parser, we preload the Q6/Q19 data into its columnar in-memory format with a single-threaded C++ CSV parser [196]. For the 311 workload, we use Weld's benchmark code, which uses Pandas to load the data. We measure pure compute time, which measures how good Tuplex's generated code is, and end-to-end runtime, which measures a realistic data analytics experience. A good result for Tuplex would show competitive compute time and an improved end-to-end runtime.

Figure 5.6 shows that Tuplex's compute time (including compilation and sampling) for the 311 data cleaning workload is within 2× of Weld's, and that end-to-end (total runtime to load the data, compile the query, and execute it), Tuplex runs the workload 2× faster than Pandas+Weld. On TPC-H Q6, Tuplex's runtime is within 2× of Weld's for Q6, despite Tuplex's lack of vectorization and its row-structured data layout in memory (Figure 5.7a), and Tuplex again outperforms Weld by 1.86× end-to-end (Figure 5.7b). Tuplex's end-to-end performance gains come from an optimization available when compiling full pipelines: instead of loading the data first and then running the aggregation, Tuplex *generates* a CSV parser and inlines the aggregation code into it. Weld, by contrast, first loads the data via Pandas to a columnar in-memory representation and then aggregates it via fast SIMD instructions. The results for Q19 are similar: due to vectorization, Weld outperforms Tuplex (without logical optimizations, 10.9s) by 2×. However,



(a) query-time only (b) end-to-end

Figure 5.7: For TPC-H Q6/19, Tuplex's generated code (without vectorization or indexes) is competitive with Weld's vectorized code and within 3–8× of Hyper's index-based execution. End-to-end, Tuplex outperforms Weld by 2× (due to its generated parser) and Hyper by 5–7× (by avoiding index creation).

Tuplex can apply logical optimizations and push down filters as Tuplex's optimizer is aware of both UDFs and the overall query structure. This awareness leads to a 3× speedup over Weld, even though Tuplex lacks vectorization.

5.1.6 | *SQL query compiler: Hyper.*

Tuplex is designed for analytics over large, non-indexed data sets. In classic SQL databases, query compilation is well-established. While Tuplex seeks to support a broader use case (Python UDFs) than SQL queries, we compare to the Hyper query compiler [119, 76] to establish a baseline for Tuplex's performance on classic SQL queries. We use Tableau's latest HyperAPI [94] (0.0.12366) to run TPC-H Q6 with 16 threads. Hyper relies on indexes for performance [113]: we expect Q6 to run an order of magnitude faster when indexes are used, as they allow to skip most of the data compared to a pure scan-based version. This comes at the upfront cost of creating the indexes, however.

Tuplex's scan-based query execution is indeed 3–8× slower than Hyper's index-based execution (Figure 5.7a). Tuplex's Python code is also more expensive to compile (120ms) than directly parsing a simple, well-structured SQL query like Q6, as Tuplex must perform additional steps like type inference and tracing. Finally, Figure 5.7b shows that Tuplex outperforms Hyper by 5–7× on end-to-end runtime, since Tuplex avoids upfront index creation and interleaves the aggregation with data loading through its generated parser.

5.1.7 | *Limitations.*

Tuplex by design cannot use some optimizations available to Weld or Hyper, because Tuplex adheres strictly to Python semantics and must forego optimizations that would violate these semantics (*e.g.*, via `-ffast-math`). Furhter, Tuplex generates code that still contains instructions to check for exceptions, while Weld and Hyper only work on perfectly clean data.

5.2 | Tuplex Performance Breakdown

The largest contributor to Tuplex's speedup over Spark and Dask is compiling Python UDFs to native code, but specific design choices improve Tuplex's performance by up to 3×.

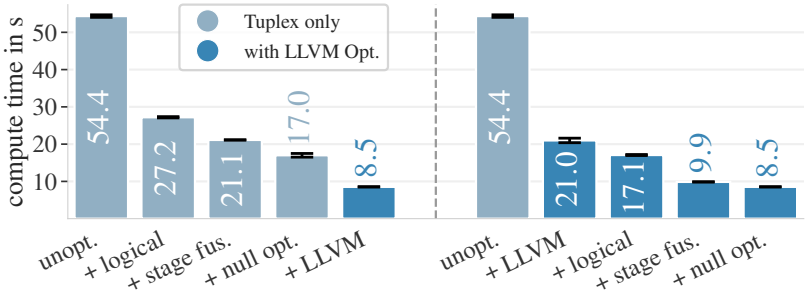


Figure 5.8: Factor analysis for the flights pipeline: Tuplex's logical and compiler optimizations help LLVM optimizers to jointly realize speedups.

We measure the impact of specific design choices and optimizations with the flights pipeline, using 4-way concurrency and with Tuplex configured to avoid swapping. Figure 5.8 summarizes the impact of each factor on **flights** (5.9 GB input data) with and without LLVM optimizers enabled, plotting times only for the compute part of the pipeline (*i.e.*, excluding I/O). There are two high-level takeaways: first, logical optimizations and stage fusion are important; and second, our optimizations give additional purchase to the LLVM optimizers. We mention results for other pipelines where relevant; these are end-to-end numbers including I/O.

5.2.1 | *Logical Optimizations.*

Tuplex compiles Python UDFs with full knowledge of their ASTs. This allows Tuplex to apply standard optimizations like filter and projection pushdowns and operator reorderings *through* UDFs—in contrast to Spark or Dask, which treat UDFs as black-boxes. We illustrate the impact such logical optimizations have with the weblogs and flight pipelines; the Zillow pipeline has few logical optimization opportunities.

In the **flights** pipeline, projection pushdown helps drop many of the 110 input columns early. Tuplex achieves a 2× speedup thanks to this logical optimization when we disable LLVM optimizers, but the benefit grows to 3× with LLVM optimizers enabled. This is caused by LLVM eliminating code that processes data eventually dropped and its ability to reorder basic blocks for inlined functions.

The **weblogs** pipeline contains a `join` with a list of malicious IPs and a `mapColumn` operator that anonymizes some records. Applying `mapColumn` to output rows of the (selective, *i.e.*, filtering) join requires anonymizing fewer rows. But Spark or Dask cannot move a UDF-applying `mapColumn` through a join, while Tuplex can, thanks to its understanding of columns read and modified in the UDF. With this optimization, Tuplex takes 27 seconds (2× faster

than the unreordered result we reported in Figure 5.3). If we manually reorder the operators in PySparkSQL, it also runs 2.5× faster (305 seconds), but remains 11.3× slower than Tuplex.

5.2.2 | *Stage Fusion.*

Systems that treat UDFs as black-box operators are unable to end-to-end optimize across them. In Spark and Dask, a UDF operator is an *optimization barrier*, while Tuplex makes stages—the unit of optimized compilation—as large as possible. To measure the impact of this design, we manually insert optimization barriers in the flights pipeline, forcing Tuplex to use additional stages. We consider Tuplex with optimization barriers that mimic Spark's optimization constraints; and Tuplex with stage fusion (*i.e.*, only the build side of a join is a barrier, cf. §4.4.5). For each, we disable and enable LLVM optimizers to measure any cross-UDF optimization enabled. Without LLVM optimizers, Tuplex takes 27.2 seconds without stage fusion and 21.1 seconds with stage fusion (22% improvement); with LLVM optimizers, runtimes drop to 17.1 and 9.9 seconds (42% improvement). Stage fusion thus enables optimization potential that improves runtime by an extra 20%.

5.2.3 | *Optional Types off the Normal Path.*

Dual mode processing allows Tuplex to optimize the normal-case path by deferring complexity to the exception-case path. We measure the impact of shifting rare null values to the general-case code path (§4.4.7). In **flights**, this optimization reduces the pipeline's compute time by 14–19%, albeit at the cost of increasing compile time by 2 seconds, which reduces end-to-end benefit. (Larger datasets would realize more of the benefit, as compile time is a constant.)

5.3 | Distributed, Scale-Out Execution

While our focus has been on the single-machine performance of our Tuplex prototype, some systems we compare to (PySpark and Dask) support distributed execution. To verify that Tuplex's performance gains are not merely a consequence of avoiding overheads associated with distributed operation, we compare these systems with Tuplex's experimental distributed execution over AWS Lambda functions.

We compare our prototype's Lambda backend with a maximum concurrency of 64 simultaneously running requests to a Spark cluster with 64 executors. We use Lambdas with 1.5 GB of memory. The Spark cluster runs on 32 `m5.large` instances that each run two executors with 1 core and 2 GB of memory per executor. This gives Spark an advantage, as it has more memory and the cluster runs continuously, while Tuplex provisions a Lambda container for each task. In addition, while Tuplex's Lambda backend writes to S3, Spark merely collects results on its driver node, as writing to S3 requires extra infrastructure [65, 191]. We run the Zillow pipeline over scaled

Setup	Spark (64 executors)	Tuplex (64 Lambdas)
100 GB	209.03 sec (σ = 10.53)	31.5 sec (σ = 8.25)
1 TB	1791.51 sec (σ = 4.38)	351.1 sec (σ = 22.10)

Table 5.3: In a distributed scale-out experiment, Tuplex's Lambda backend outperforms a Spark cluster by 5.1–6.6×.

datasets of 100 GB and 1 TB, with data stored in 256 MB chunks in AWS S3. To verify that the compute speed of `m5.large` VMs is comparable to 1.5 GB Lambda functions, we ran a microbenchmark over one 256MB chunk. It takes 3.99 seconds on an `m5.large` VM, while our code within a Lambda function takes 4.00 seconds on average, with some variance (min: 3.68 sec, max 9.99 sec).

Table 5.3 shows the results. For Spark, we show numbers for the tuple-based pipeline; the dictionary and SparkSQL versions are 10–20% slower. Tuplex completes the pipeline in 31.5 and 351 seconds for 100 GB and 1 TB, 5.1× and 6.6× faster, respectively, than the fastest Spark setup. This difference comes from Tuplex's compute speed, which outweighs the overheads associated with Lambdas (HTTP request, queueing, container provisioning, etc.). In terms of direct monetary cost, Tuplex is competitive at 4¢ for 100 GB (Spark: 3.7¢) and 55¢ for 1 TB (Spark: 32¢), while also avoiding the setup and provisioning time costs, idle costs, and EBS storage costs that Spark incurs on top of the EC2 VM costs. This suggests that Tuplex can be competitive both on a single server and in scale-out settings.

5.4 | Discussion and Experience

Tuplex's primary objective is high performance for pipelines that include Python UDFs. But the dual-mode execution model may also help Tuplex users avoid some long-standing challenges of pipeline development and debugging [75, 131]. Key to this is Tuplex's guarantee that pipelines never fail because of malformed input rows: instead, Tuplex does its best to complete the pipeline on valid, normal-case rows and reports statistics about failed rows to the user. It is difficult to quantify the impact of failure-free pipelines on developer productivity. However, in our anecdotal experience implementing pipelines we found Tuplex preferable for several reasons:

1. Although our evaluation data sets are fairly "clean", they contain a small number of anomalous rows, which often caused hard-to-debug failures in Spark and Dask.
2. Representing rows as tuples instead of dictionaries improves PySpark performance, but the numerical indexing took painstaking work to get right. Tuplex avoids the speed-usability tradeoffs and has the same performance for tuples and dictionaries.
3. Making null values work with Dask/Pandas required using special datatypes (*e.g.*, `np.int64`), rather native Python types, as Pandas fails on `None` values.
4. The semantics of special-purpose operators designed to help developers avoid UDFs differ from Python code.

For example, SparkSQL's `regex_extract` returns an empty string when there is no match, rather than `NULL` as a Python user might expect (Python's `re` returns `None` in this case). The weblog dataset has two anomalous rows, which cause SparkSQL to silently return incorrect results, while Tuplex correctly reported them.

5. We compared to Weld using the Pandas cookbook's sub-sampled 311 dataset [34] (99k rows) scaled 2,000× in §5.1.5, but Tuplex works out-of-the-box on the full NYC 311 dataset [122] (22.4M rows), while Weld, PySpark, PySparkSQL, and Dask all fail and require changes to the UDF code for the realistic dataset.

We spent substantial time tracking down edge cases in framework documentation for other systems, while Tuplex's Python UDFs behaved as expected. We also found that Tuplex's reporting of exceptions and failed rows helped us track down bugs in our pipelines.

Tuplex's dual mode processing requires a representative sample. Like with any sampling approach, an unrepresentative sample can lead Tuplex to deduce an incorrect common case. If the sample itself produces only exceptions, Tuplex warns the user either to revise the pipeline or increase the sample size.

6 Viton

With Tuplex, we demonstrated that is possible to provide for the specialized problem of large-scale processing a more efficient approach than writing a data pipeline in standard Python or using popular data processing frameworks. A key reason for the slowness of Python code as outlined in chapter 2 is that it is conservative and makes no assumptions about the schema of the data processed: for example, columns may contain arbitrarily-typed data and dictionaries might have arbitrary structure. This induces substantial overhead, as the executing machine code contains many checks and extra instructions that are unnecessary in the common case. With dual-case processing we presented a solution to this efficiency problem by *specializing* the executing machine code to the input dataset and its common-case structure. However, the performance of such specialized code fundamentally depends on the quality of the sample.

In our prior, work we approached sampling in the same way that a traditional, centralized database query planner does, and used a small sample to infer a *global* common-case to specialize to [172]. This is simple, and works well if the sample's distribution matches the overall data distribution. But the structure of real-world datasets—particularly those collected over a long time period—often changes over time, and data distributions shift. This heterogeneity can occur due to schema changes (*e.g.*, Github's API schema has changed 417 times since 2017 [45]), due to changes in the data (*e.g.*, the COVID-19 pandemic induced a rapid shift in the BTS flights dataset [185]), or due to chronological structure (*e.g.*, a file that contains all records from a given month will have constants in the `year` and `month` fields). Simple sampling strategies may fail to pick up such changes, and consequently may lead to incorrect query plans and poor code specialization. But more complex sampling strategies also cannot solve the problem. As we show in this chapter of the thesis, sampling the dataset in more complex ways can in fact eliminate specialization opportunities as the sample's distribution becomes too noisy, and sampling many parts of the input introduces substantial runtime overhead. This problem exacerbates in a distributed setting, where data is large and likely exhibits some heterogeneity.

Hyperspecialization: We therefore motivate *hyperspecialized* data processing, which composes coarse-grained sampling, query compilation, and modern parallel serverless infrastructure into an efficient approach to processing heterogeneous datasets. The key idea is to generate maximally-specialized native code by *fitting the code to the data* in a more fine-grained way: each parallel worker samples its particular input, and then generates its own, hyperspecialized code to process it, tightly fitting the common case. This maximizes CPU efficiency, particularly

```python
# `features` is a set of parsed column values
def impute_values(year, month, delays, features):
    arr_delay = delays['arrival_delay']
    if year == 2003 and month < 6 or year < 2003:
        if arr_delay <= 0.:
            return {'nas_delay': None, ...}
        elif arr_delay < 5.:
            return {..., 'carrier_delay': arr_delay, ...}
        else:
            delays = model.predict(features)
            return dict(zip(factor_names, delays))
    else:
       return delays
```

Listing 8: A UDF that imputes missing values for delay factors for months prior to June, 2003. A machine learning model is used to impute missing values.

on large datasets with shifting patterns and marginal distributions. We expect that hyperspecialization will improve efficiency at no cost to the developer: it reduces the resources consumed by data analytics computations while doing the same work. The key reason is that serverless computing and specialized code generation can hit a sweet spot where the high parallelism of parallel processing begets additional specialization opportunities, as the sampling and specialization become more fine-grained, which further improves the efficiency of the generated code.

6.1 | Motivation

We motivate hyperspecialization with a running example of a typical data science task: cleaning up data about flight delays [185] which is the same dataset used previously in chapter 5. The dataset covers 34 years of flights, and some schema evolution naturally occurred. One schema change is the breaking up of the flight arrival delay in minutes into multiple factors [186]: the dataset has `NULL` values for all newly-introduced columns for any dates before the schema change in June 2003. Imagine a data scientist who seeks to impute the missing values for the delay factors using a suitable machine learning model for the old records. Given the rather complicated semantics of the missing values and their correlation with other columns (*e.g.*, arrival delays are only given for flights which were not diverted, cancelled, or arrived early), the user decides to write the imputation routine as a single function in Python(cf. listing 8): Traditional systems like Spark run this Python code as a user-defined function (UDF), which they treat as a "black box" and cannot optimize via classic query planning and logical optimization.

However, observe that each individual input file, which represents one month's worth of flights, will run through a particular branch of the outermost `if` statement. Only months before June 2003 require imputation via the model and enter the `if` branch; all others follow the simpler `else` branch. The model's features passed as argument

`features` consists of data obtained from many columns, but `features` is unused in the `else` branch. Thus, parsing the columns passed in `features` is unnecessary for input files after June 2003, and the parsing code for these columns could be eliminated. But this high-impact optimization is out of reach both for classic query optimizers and for a UDF compiler: a query optimizer cannot reason about the Python UDF code (which it treats as a black box), and a UDF compiler can only reason statically about the code or about the input data as a whole if provided with a sample.

Savvy users might consider to utilize some clever rewriting (*e.g.*, splitting the UDF into multiple UDFs) and performance-motivated mix-in of SQL expressions (to allow query optimizers to work better) to improve the execution speed. But these optimizations require expert knowledge and careful tuning.

With *hyperspecialization*, we propose an automated technique that can exploit optimization opportunities that require reasoning about local properties of parts of typical semi-structured datasets. Hyperspecialization samples the input data and generates UDF code on a fine-grained, per-input file basis.

6.2 | Overview

A traditional DBMS stores statistics that help in query planning—*e.g.*, data types, statistics, and locality information—centrally and keeps them updated. Such meta-data allows a query planner to produce an efficient query. In addition a DBMS is free to (re)organize data to support repeated queries most efficiently. This can be either done through indices or physical reorganization [59]. Specifically for semi-structured data in the form of JSON files, breaking them up into column chunks has been proposed to deal with heterogeneity and schema evolution over time [31]. But pipelines over raw, non-relational data stored in CSV or binary format have no such comprehensive information, so they must sample the data directly and assume that the sample is representative. This introduces a trade-off between more complex sampling—which increases the likelihood of obtaining a representative sample—against planning and execution speed. Sampling the whole dataset can be prohibitive, as it may require loading large amounts of data from cloud storage and parsing them. But a more limited sample (*e.g.*, the first input file only) may lead the query planner into a suboptimal plan. This risk increases when data sets grow over time, as schema changes and the marginal distribution shifts, *i.e.*, the dataset becomes more heterogeneous. Existing systems to process raw-data typically assume a fixed schema while scanning [3], or require prior ingestion to detect offsets over raw-files before allowing users to issue queries [199]. In this work, we focus on a scenario which have no previous knowledge of the data, i.e. all sampling and inference has to be performed on-the-fly while processing the query, which is typical for large-scale ETL jobs. To demonstrate our idea, we built on top of Tuplex and extend its incomplete AWS Lambda backend.

Tuplex's specialization strategy works well if the common-case assumptions remain static across the whole dataset and most records are processed on the fast path. But if the dataset is heterogeneous or Tuplex infers the wrong common

case, many records will require processing in the interpreter, and Tuplex's performance degrades to that of a Python interpreter. To show this shortcoming of Tuplex, we motivate our work on hyperspecialization through an experiment.

6.3 | Don't just sample globally!

In this motivational experiment, we compare the effect of different sampling strategies on the specialized code Tuplex generates for the example pipeline described in §6.1. We use Tuplex as a case study here, but investigate more generally what specialization opportunities different sampling strategies can help discover, and the results should generalize to other systems that adapt based on sampling the input data.

Tuplex deduces types for the slow and fast path using a small sample of the input data (256 KB by default). To keep sampling costs low, Tuplex's original approach samples just the first rows of the first input file. We compare

sampling mode	(a)	(b)	(c)	(d)	(e)	(f)
first rows	x	x	x	x	x	x
last rows		x		x		x
first file	x	x	x	x		
last file			x	x		
all files					x	x

Figure 6.1: We compare six different sampling modes; Tuplex's default mode samples the first rows of the first file (setup (a)) and the limits the total number of rows to sample to 10,000.

this approach to five other sampling strategies (Figure 6.1) using the query described in §1. The different strategies

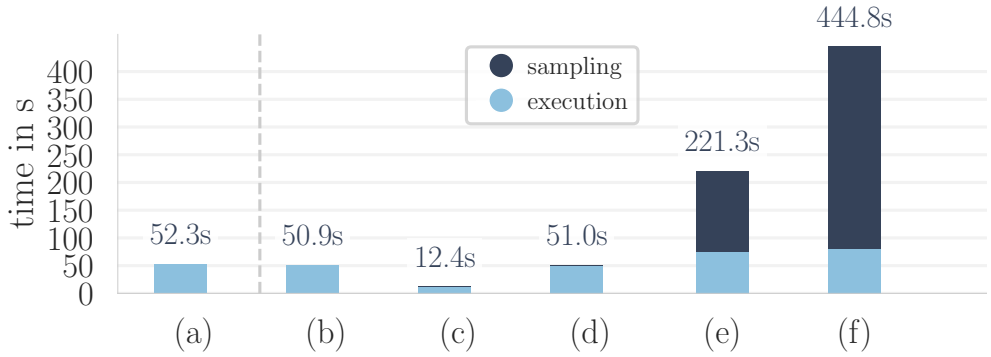


Figure 6.2: The performance of Tuplex varies widely depending on the sampling mode selected (cf. table 6.1). Here, sampling mode (c) which takes the first rows of the first and last file performs best, as it by chance detects a performant schema for the input data.

vary combinations of which input files Tuplex samples (first file, last file, or all files), and which rows in these files it samples (first or last rows). For each strategy, we measure the sampling time and pipeline execution time that results from the column types ("schema") that Tuplex detects based on the sample. All experiments use Python 3.9.8 and Tuplex's AWS Lambda backend, which runs each parallel processing task in a serverless function that we configure

with 10 GB of memory. All queries are invoked from an AWS EC2 r5d.xlarge instance in the us-east-1 zone. We repeat each experiment ten times on hot AWS Lambda functions, and runtimes are end-to-end time averages recorded on the EC2 driver machine over 10 runs each. Figure 6.2 shows the results which exhibit vast differences in runtime depending on which sampling strategy is used.

Finding the right sampling strategy: Un-surprisingly, increasing the amount of data sampled increases the sampling time for modes (e) and (f), which sample all 410 input files. The problem here is that for data stored in S3 one or more requests have to be issued to retrieve data samples. We use multi-threading (8 threads) to control S3 request cost, but the numbers still suggest that sampling every single file is not a good idea as it does not produce a better code path performance-wise for this particular query than if only a single or two files were sampled, as fig. 6.2 shows. Sampling modes *A*, *B* and *C* sample different files and file parts and result in similar performance, whereas *C* has the best performance and produces a code path that runs $\approx 4\times$ faster than any of the other modes. But it is not clear to a user upfront which sampling mode will perform well, or which files to choose to sample from for a particular query. A more robust approach that automatically detects the right sampling strategy would help to deliver more reliable runtimes. This experiment illustrates that determining the right sampling strategy is difficult, and that a global sampling strategy can result in missed code specialization opportunities.

6.4 | Hyperspecialization

The central idea of hyperspecialization is to not rely on a single, perfect global specialization setting but rather perform on-the-fly *reoptimization* while executing a query. To achieve this, we divide the execution of a query into two steps when it comes to planning: In a first step, which we execute on the client, we draw a small initial sample to estimate an initial schema for the query to perform initial query planning like detecting which stages to generate. We also generate and compile a general, compiled code path that is globally optimized. Each stage is then executed using a varying degree of Lambda executors. With hyper-specialization mode active, the initial batch of Lambdas is assigned a specialization unit. While there may be different strategies on how to identify and assign specialization units, we decided to go with a simple approach: each file serves as a specialization unit. We base this on the assumption that likely there has been an initial partitioning of data performed along some attribute is when attempting to processing multiple files in parallel. In a sense, individual files marginalize the data distribution such that marginal distributions have overall lower variance. For historical data, this is typically the time of collection but other schemes may be possible.

1.	Deserialization / Parsing
2.	Sampling
3.	Specialization (schema detection, logical re-optimization)
4.	LLVM IR generation
5.	LLVM IR optimization
6.	IR to x86 lowering

Figure 6.3: Generating specialized code for a unit incurs various costs due to the different steps involved when producing a new code-path.

Balancing optimization cost: On our quest to design a new distributed system that supports compiling Python to efficient code, we have to balance carefully where to generate, optimize and execute which code. Typically, the driver machine issuing the query to each Lambda executor has limited parallelism available and a slow connection to any data stored in a blob service like S3. However, we consider any compute time spent on the driver machine to be essentially free, whereas every single millisecond we have to spent on Lambdas doing not actually work multiplies by the parallelism we employ quickly. Keeping overheads low on each Lambda is crucial. But, spending too much time on the driver to generate, optimize and run code overall results in a slow query and bad user experience. Therefore, we find a compromise: we perform a raw, cheap global optimization using a cheap sample on the driver that we use to split a query into stages, project columns initially and perform filter optimizations (like pushing filters through joins). We then expect re-optimization on Lambdas to resolve any initial sampling error we may have incurred on the driver and also address heterogeneity within the input data. With this design choice we balance the cost of too much optimization and code generation on a Lambda versus increased end-to-end query time. For datasets that are homogeneous, we allow users to disable hyperspecialization for lower execution cost. But for heterogeneous datasets hyperspecialization allows to keep sampling costs on the driver low, and resolve sampling errors on each Lambda by re-optimizing the code during query execution. However, making hyperspecialization work in practice is challenging.

6.4.1 | *Challenges*

The overall challenge of hyperspecialization is that any cost to perform hyperspecialization must be traded off against the performance benefit better fitted code brings. In particular, we would want to avoid a situation where performing hyperspecialization would perform worse than if just the global normal, global or fallback (interpreter) code path was used. The cost to specialize code for a unit (file) can be broken down into multiple steps which comprise sampling, specialization and compilation cost, as shown in figure 6.3. In order to generate a new specialized code-path on a Lambda, any operator that forms a stage together with the code of their respective UDFs has to be shipped to an executor to specialize on. Consequently, we ship code in the form of ASTs with annotations from the client

and serialize additional information using Cereal [49] which is an efficient library to serialize C++ structures, which allows us to keep serialization costs of a stage low. Next, a new input data sample is drawn for the unit. Controlling the sampling cost here is challenging, as we want on the one side to avoid issuing too many S3 requests but also ensure that the sample is representative. For this reason, we issue two requests to S3 to get a block of fixed size of the start and end of a file to base the initial sample on. To further reduce sampling cost, we use stratified sampling instead of parsing all available rows in the received data blocks. With stratified sampling we partition the input data into groups (strata) of equal size, and then draw randomly from each group an identical amount of samples. With this approach we can avoid sampling errors for e.g. sorted data where using randomized sampling or sampling the first and last rows like in Tuplex produces sampling errors. After obtaining a new batch of sample rows, Viton performs type inference to detect whether a differing schema is present. In the case of a differing schema, the complete stage is retyped. With this retyping, both logic and data representation can be more tightly fitted to the concrete input data the Lambda is about to process. Moreover, specializing on the new sample yields another opportunity: the stage can be logically re-optimized, i.e., it could happen that particular code-paths for this sample do not require to parse the same input columns as in the original logical pipeline that has been globally optimized. The same applies to operators potentially becoming dead-code, or a better reordering possibility for filter operators. In a sense, the specialization step here combines *logical* with *compiler* optimizations. In a next step, the optimized stage is compiled to LLVM's intermediate representation (IR). This allows to use LLVM's infrastructure to perform another optimization step using LLVM's optimization passes before emitting machine code. In order to make hyperspecialization viable, the cost of executing all of these steps has to be low enough to be offset by a performance gain high through a more efficient code-path for the normal case. For this reason, Viton uses more aggressive optimizations compared to Tuplex which may work for a tiny subset but may likely fail globally. Viton thus optimizes most aggressively *locally* and takes a less aggressive approach *globally*. We reflect the same design choice by using an aggressively configured LLVM JIT compiler for the normal-case, and a less aggressive optimizing LLVM JIT compiler for the general-case code path in order to control compilation costs on Lambdas. Compared to Tuplex, which uses LLVM passes similar to clang's `-O2`, we disable most LLVM optimization passes in Viton on the Lambda executors as we want to avoid Lambdas to spend too much time on code optimization.

6.5 | Optimizations

To generate more narrowly fitted code to specific inputs, we built upon both logical optimizations and compiler optimizations Tuplex employs such as filter pushdown, breakup, reordering and column selection/projection pushdown and introduce two new optimizations:

6.5.1 | *Constant-folding*

Tuplex speculates on whether types can be NULL or not. We demonstrated [172], that this particular optimization on its own can improve performance by 15%. For Viton, we expanded this optimization by speculating on the constness of arbitrary values. This means, instead of emitting a check on whether a column is potentially null we emit a check whether a value is of a particular constant or not. This means, that if a column is considered constant it does not need to

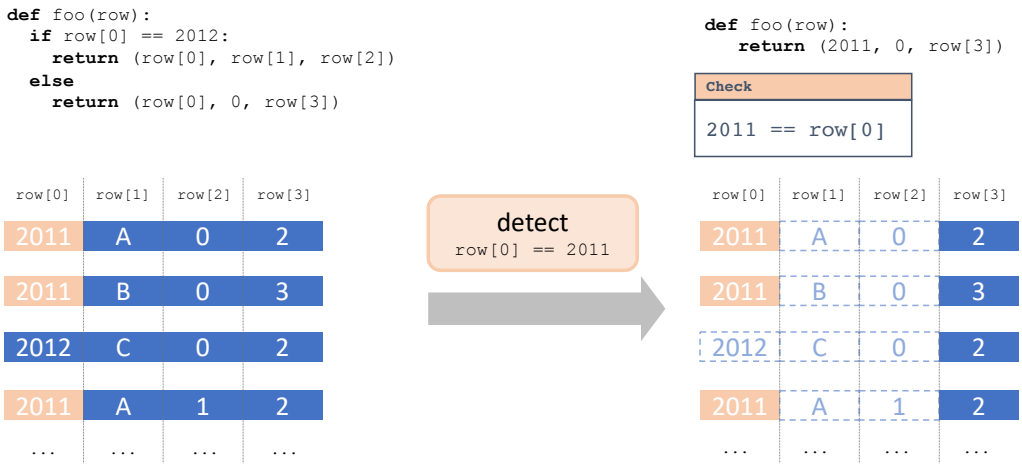


Figure 6.4: Constant-folding allows to simplify UDFs and enables further logical optimizations. Viton detects for the sample that in the normal case the first column — indexed via `row[0]` – is mostly `2011`. By treating the first column as a constant of value 2011, Viton can eliminate the if-branch in the UDF. Reapplying Viton's logical query optimizations allows to project the second and third column out. However, constant-folding requires to emit an additional check as part of the row classifier (cf. §4.2).

be materialized anymore in the normal-case and the UDF compiler is able to avoid costly operations by folding value accessed with a constant instead. Moreover, in the best case this allows to collapse branches or even whole UDFs to a constant value. From this standpoint, constant-folding improves the execution speed for UDFs. But, there is an added benefit: Constant-folding UDFs may enable logical optimizations like filter pushdown, column projection or dead-code elimination to better optimize a pipeline altogether. To illustrate this better, let's take a look at a concrete example as depicted in figure 6.4. Here, a sample is given of which the majority of rows make it seem likely that the first column is always 2011. Thus, a check to check for the value 2011 (and also with this that the first column is of `i64` type, and not-null) is added when determining whether the row should be processed on the normal-case or not. However, knowing in the normal-case that the first column is always 2011 allows to evaluate the branch condition `row[0] == 2012` to `False` which makes the "then" branch dead-code. But if the logical optimizer is now rerun over the pipeline, it can detect that for this UDF parsing columns 1, 2 is not necessary anymore (the first column still needs to be parsed to perform the check). In particular this means that as input to the UDF neither column 0,1 or 2 need to be materialized

anymore. In the case the check fails (as for the row `(2012, A, 1, 2)` shown in the example) the execution of the UDF is deferred to the general-case code-path (which may deferred it to the interpreter as well, as shown in fig. 4.1).

To implement constant-folding within UDFs, we leverage Tuplex's type system by introducing a new concept of *optimized types*. Instead of a rather complex analysis of control flow, using types allows to re-use the existing typing infrastructure. The idea is to basically create a specialized subtype of an existing type within Tuplex, that can at any time be deoptimized to the original type. I.e., if a parameter `x` is assigned an optimized type `_Constant[i64,2011]` it can be deoptimized to an integer `i64` whenever needed. Folding both expressions and statements based on constant types allows to optimize ASTs before generating LLVM IR out of them. Instead of checking for data-flow, it's sufficient to check for and combine constant types.

6.5.2 | *Filter promotion*

When writing pipelines that involve filter operators, users often write the logic within other operators specifically to match only rows that pass a predicate. E.g., imagine a data scientist analyzing Github events given in the form of semi-structured JSON data. Frequently, in semi-structured datasets the structure differs between different subgroups, i.e. a `PushEvent` may be vastly different than a `WatchEvent`. Detecting these different schemas within a dataset is challenging for traditional analytics systems. For this reason, in systems based on SQL usually only primitives that perform explicit parsing are implemented like `JSON_EXTRACT` that expect strings and user-logic to perform explicit type casting. In the case of Python however, there is a natural mapping of JSON to Python dictionaries which makes writing explicit type cast and parsing logic unnecessary and cumbersome. Indeed, other approaches like using a dedicated JSON query language (jsoniq [37]) have been proposed to overcome the challenge of supporting data preparation pipelines early enough for nested and heterogeneous datasets which has been described as a primary challenge in [42] [42]. With *filter promotion* we want to automate the implicit behavior of users writing their logic, while overcoming the obstacles that prevent traditional systems to optimize queries over semi-structured data better.

The core idea of filter promotion is to apply the filter (if cheap enough and not dependent on other operators) on the initial sample given, and if not all rows from the filter are filtered out to promote the filter to become a normal-case check. We illustrate this in figure 6.5, where the schema according to the blue dots would dominate without filter promotion. But for the schema corresponding to the blue dots, the filter predicate may evaluate to `False`, which would stop processing rows further down the pipeline. This would translate to a misoptimization, and may even result in a majority of rows causing problems in the normal-case as a wrong schema is assumed by the framework – but not by the user. Applying filter promotion would promote the UDF of the filter operator to become part of the row classifier as a normal case check and use the filtered sample to propagate types and perform

tracing through the rest of the operator graph. This means that for all subsequent operators of the promoted filter operator, a *more restrictive* schema may be assumed which is informed by the filtered sample. Restricting the sample may be beneficial, as for the filtered sample some fields may become constant. More concretely, when a user would for example like to perform computations on `ForkEvents` which are much rarer than `PushEvents` Viton is able with filter-promotion to assume the schema of `ForkEvents` and can subsequently avoid parsing unnecessary fields. In case the filter operator produces an empty sample, Viton avoids applying the filter-promotion optimization.

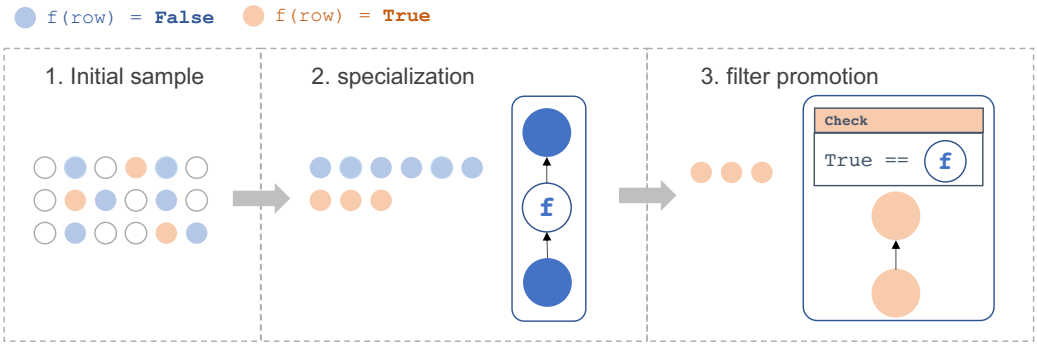


Figure 6.5: Filter promotion manipulates the sample used by promoting a filter to become a check for the normal-case and allows schema decision and logical optimizations to be based on the filtered sample instead of the initial one.

6.6 | Implementation

To demonstrate the viability of our idea to hyperspecialize on heterogeneous datasets, we created Viton, a prototype hyperspecializing compiler based on Tuplex. Creating Viton required adding support for the two optimizations described in §6.5 and extending the early-stage Lambda backend of Tuplex to support shipping stages in the form of ASTs to Lambda executors. For this, we implemented a custom AWS Lambda runtime as this was more efficient in micro-benchmarks than building on top of existing runtimes in AWS Lambda. In addition to implementing per-Lambda, per-input file sampling and hyperspecialized code generation, Viton also adds support for semi-structured JSON files with a parser built on top of simdjson [85].

Redesigned architecture: To apply hyperspecialization on individual Lambda invocations, Viton follows an architecture that moves code generation onto Lambdas (Figure 6.6). Users still write queries using standard Python, just like in Tuplex. The driver performs an initial sample to inform basic query planning (logical optimizations, slow path code generation). Viton serializes a full stage with all its UDFs as ASTs, including lightweight annotations from the driver based on the initial sample, and transfers it on each invocation. In contrast to Tuplex, Viton then generates and compiles an individual fast path on each Lambda function after drawing a sample from the Lambda function's input to further

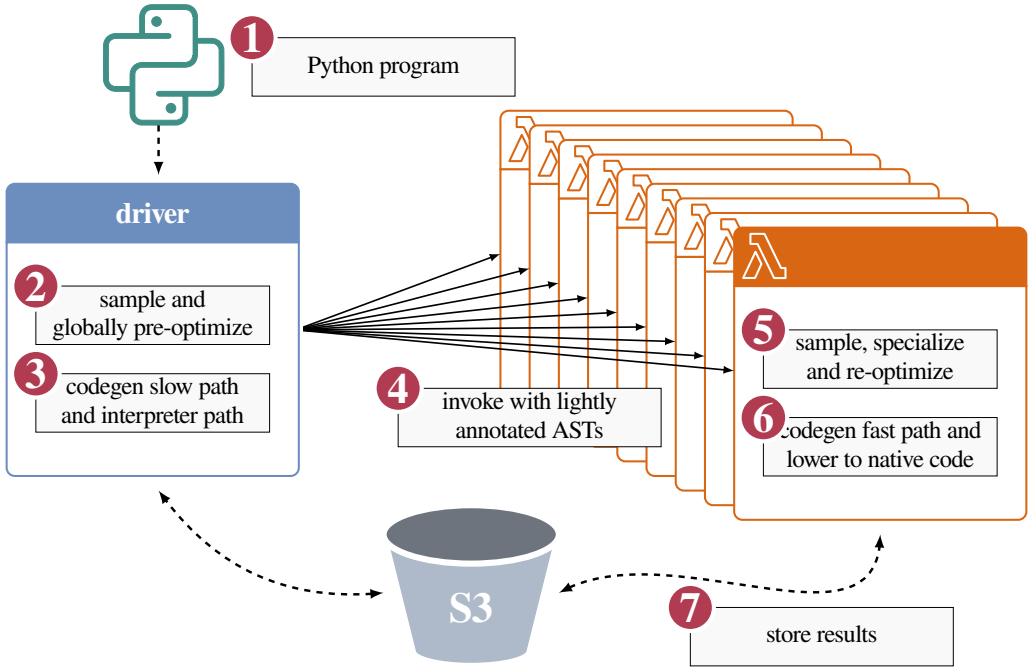


Figure 6.6: Viton system architecture: the driver performs initial sampling and code generation, but each serverless Lambda function further samples and specializes to its particular input.

specialize the AST. If deoptimization occurs—*i.e.*, the slow path or interpreter execution is required—the Lambda runs the slow path LLVM IR obtained from the driver or the fallback code path via an embedded CPython interpreter.

Opportunistic compilation: When using hyperspecialization together with aggressive optimizations, some rows will likely require processing through the slow compiled path based on the general case schema. Tuplex always compiles both the fast and slow path in parallel. Yet, this may be undesirable in Viton on each Lambda as processing will only start once compilation of both paths finished. For complex schemas involving large structs (like for Github events) compile time for the general case may be significant. Viton therefore compiles the general case code path in parallel while hyperspecialization is performed. But this comes at the cost of using the CPU of the Lambda for compiling a path that may not be needed if all rows process through the normal case and no deoptimization is triggered. For this reason, Viton provides users with the option to enable or disable opportunistic compilation of the general case path. When opportunistic compilation is disabled, compilation of the general case path is only performed when rows actually need to be processed through it. This allows to balance whether users speculate on a general-code path likely being necessary or whether they only want to pay for the compilation cost if it actually is needed. Anecdotally, we observed that opportunistic compilation slightly increases query cost, but leads to overall lower end-to-end runtime when enabled.

7 Viton – Evaluation

In this chapter, we evaluate the benefits of hyperspecialization using two different queries two over different datasets and seek to answer the following questions:

1. To which extend does hyperspecialization reduce query runtime and cost?
2. How does hyperspecialization compare to other, global sampling strategies?
3. Where do the benefits of hyperspecialization stem from?
4. How does Viton compare to other, state-of-the-art serverless frameworks?

Experimental Setup: We configure each Lambda to run a single Viton executor that uses up to 10 GB of memory (512MB temp storage) and a maximum of three threads. As of May 2023, a Lambda instance with 10 GB of memory has six vCPUs, three of which we use for processing and three for S3. We run the client on a single `r5d.xlarge` EC2 instance.

7.1 | Flights query

In this experiment, we perform a data cleaning query that imputes missing values for delay factors for years and months prior to 06/2003 and retrieve the cleaned result for years 2002 to 2005. The input dataset [185] consists of 410 files (total: 83.51 GB) with varying size from 177MB to 284 MB with each file containing the full data for a single month ranging from 10/1987 to 11/2021. In hyperspecialization mode, we specialize code for each file, i.e. there are in total 410 specialization units. To control sampling cost, we use stratified sampling using ten samples (without replacement) over a strata of 1024 rows. The client samples the first and last 32MB of each file (and the Lambdas as well), which results in an average of $144-158$ samples per file. The motivation to use stratified sampling is to control sampling cost. Because the files are sorted by date and carrier, simply using the first n rows would lead to sampling errors that can be avoided by using appropriate strata. Reducing the amount of samples allows to achieve sampling times of 6.3–7.1ms/row per lambda (4.1–8.9ms/row on the client) which in total translates to spending $\approx$ 1s per lambda (0.6–1.4s on the client) on sampling. Opportunistic compilation is active for both global and hyper scenarios (benefiting the global scenario overall).

When enabling hyper-specialization, in our first experiment we compare global specialization vs. hyper-specialization. When each Lambda specializes to its concrete input schema, we can achieve a speedup of 2.05x.

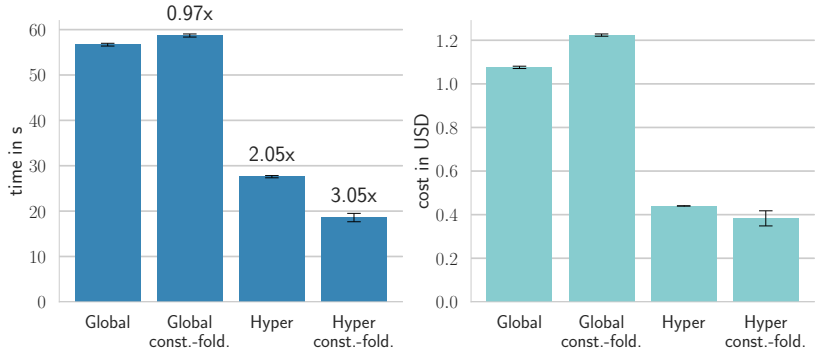


Figure 7.1: Hyperspecialization allows 2–3× improvement of end-to-end performance improvement and cost reduction by a factor of ≈3×. Applying constant-folding globally causes misoptimizations, as wrongly the fourth quarter is assumed to be constant.

Activating more aggressively specializing optimizations like constant-folding causes a drop in performance in the global scenario: on the client the sample wrongly detects the column `'quarter'` to be constantly 4. While there is a benefit for files which fall into the last quarter of a year, this also causes the check to fail in the normal-case for all other quarters. The runtime benefit of the specialized files does not outweigh the cost of failing normal-case checks which leads to an overall degradation of end-to-end performance by 4%. On the other hand, aggressive specialization allows files to simplify their pipeline for multiple years and improves performance to a factor of 3.05× (cf. figure 7.1). Looking at which code-paths individual rows take for each setting, we can explain the results more precisely. For the global specialization, we see that a significant number of rows still get processed via the general or fallback code-path (cf. figure 7.2). When applying hyperspecialization to 202,687,655 total input rows, 0.1% less rows will require the

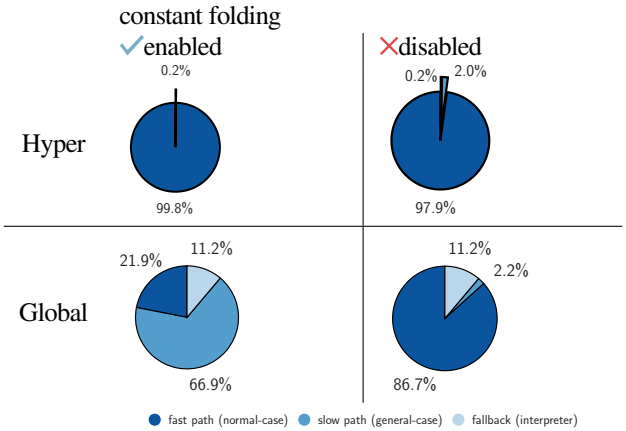


Figure 7.2: Breakdown of which code-paths input rows took in Viton. When globally specializing the code, applying constant-folding leads to misoptimization and places 66.9% unnecessarily on the general case code path. Hyperspecialization is able to fit the code better to individual subsets of the input data, and constant-folding results in only 0.2% of rows requiring still processing in the general case code path.

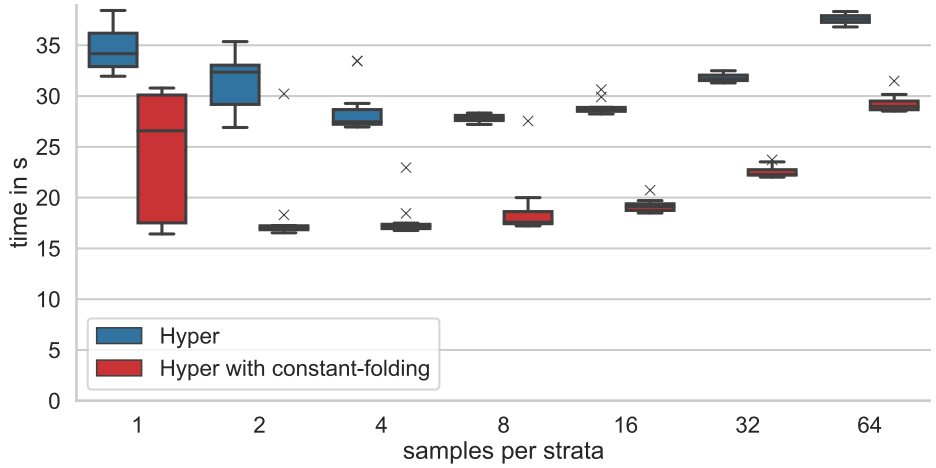


Figure 7.3: Hyperspecialization is sensitive to the sample quality obtained on each Lambda. Increasing the number of samples per strata (size=1024) helps to lower variance, but increases overall compute time due to increased sampling cost.

general case, and 10.9% less the fallback. As a result 11.1% more rows can be processed via the normal case. The normal case itself also becomes more efficient. On the contrary, using aggressive optimizations like constant-folding globally leads to execution becoming dominated by the general-case as the normal-case checks fail for a majority of files. However, for sufficiently homogeneous subsets, such optimization can yield a boost. In hyperspecialization mode with constant-folding enabled, merely 0.2% of all rows can not be processed using the normal case for the flights query.

7.1.1 | *Stratified sampling*

The stratified sampling strategy we use to control sampling cost comes with a drawback of introducing a lot of variance because the sample may not be representative enough. E.g., when taking a single sample per strata, hyperspecialization picks a wide range of possible specialization scenarios as no schema dominates clearly. Increasing the number of samples per strata (without replacement), allows to reduce the variance at increased sampling cost. Thus, sampling cost must be carefully balanced against sampling accuracy and specialization opportunity. We show the sensitivity of hyperspecialization towards sampling accuracy in Figure 7.3: for this experiment, we vary the number of samples taken for each strata. Un-surprisingly, using a single sample per strata results in a high variance for the end-to-end performance as the aggressive optimizations in hyper-mode for each Lambda may pick a sub-optimal schema and setting. While the overall end-to-end time is impacted by sampling time, using more samples reduces the variance. Based on these findings, we chose to pick 10 samples with a 1024 strata size as a good compromise to safe-guard against high sample variance and still avoid sampling cost to dominate overall query execution for our initial result shown in Figure 7.1. When sampling using all rows for the first and last 32MB of each file, the processing time in hyper-mode with constant-folding enabled approaches 55s.

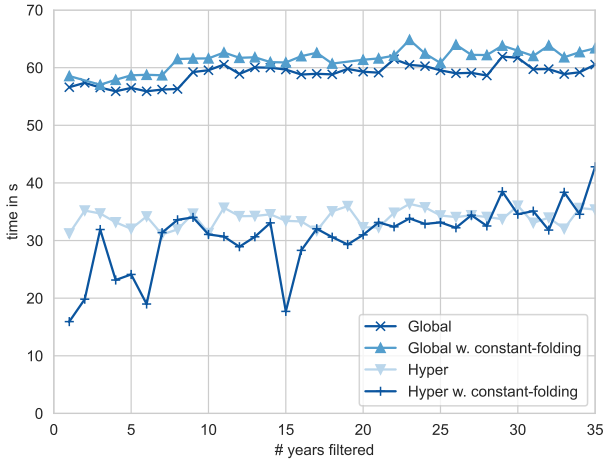


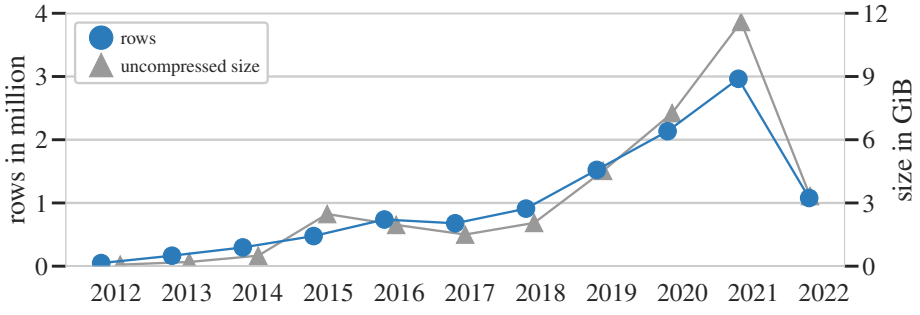
Figure 7.4: Median of 5 runs for varying filter predicate. Outliers for hyper are due to the sensitivity of the hyperspecialization with respect to to the sample and stragglers as Viton has no straggler mitigation strategy yet.

7.1.2 | *Varying the filter predicate*

A curious reader may wonder whether the primary reason for hyperspecialization to work is its ability to automatically exclude files based on constant-folding the filter in the flights query. Indeed, why not simply use a glob pattern to only process relevant files for the query? Certainly, this is possible but it places the burden on the user to clearly understand which files to process and how to process each file upfront. Viton on the other hand *automatically* detects files that can be largely ignored during processing. The core idea of Viton is to detect which columns are likely to be constant and then specialize the pipeline for the normal-case in this case. As a result for the flights query, Viton detects for files that correspond to the years 1987-2001 and 2006-2021 that in the normal-case it's sufficient to execute the normal-case check on month/year and then evaluate a collapsed pipeline that only consists of the filter operator (as it evaluates to a constant false then, which allows to drop subsequent operators). But, this raises the question of how much of the speedup is actually due to this smart filtering strategy and how much can be attributed to better capturing the schema per file. To break the effect of constant-folding the filter predicate, we modify the flights query by varying the filter predicate to span between 1-35 years, always including the year 2003 where the schema change happened. Figure 7.4 shows the results. When decreasing the filter selectivity, more rows get processed which leads to an overall end-to-end runtime increase. Effectively, hyper-specialization still outperforms global processing as it allows to (mostly in the case of non-constant folding) avoid the costly interpreter path (cf. Figure 7.2).

7.2 | Github query

In a second query, we use historical data collected from Github containing raw information about 20+ different events which has been continuously collected since February, 2011 [50]. Due to resource constraints, we limit out



][h!]

Figure 7.5: A natural trend of APIs is to "fatten" up over time as more and more fields are introduced. This trend also applies to the Github archive dataset, where the number of rows and respective file sizes shown here for the 15th October of each year from 2011–2021 increases till a reduction in 2022.

experimental dataset to a subset of 11 files for October 15th of each year (35.5GB total dataset size). Within the GH archive dataset, data is organized as newline-delimited JSON files for each day. Due to the growth of Github over the last decade, schema changes are frequent (3,748 on 417 days [45]), e.g. like the introduction of stars [118] that gave the old field `watchers` the meaning of the new field `stargazers` but kept the field with new semantics. In addition, the historic data has been also collected using different APIs: From 2/12/2011 - 12/31/2014, a now-deprecated API (Timeline API) was used, while later data uses the Events API. Each row in the dataset shares common fields, but names where information regarding the repository affected by an event may be either under a key `'repo'` or `'repository'` depending on the year. Because of the frequent changes to the API, and in order to not lose any information, any fields that can not automatically extracted to fit the predefined schema are dumped into a `'payload'` field. The structure of the `'payload'` field thereby heavily varies depending on event-type and time of collection. Over time, there is a trend of the API gaining additional fields which increases schema complexity and file-size (cf. fig. 7.5). This makes it nearly impossible for a data-analytics framework to define a global matching schema for the dataset. As a result, a user is required to write either: 1. dedicated logic for each single file and union the results, or 2. write custom extraction logic and ensure compatibility across making it impossible for a data-analytics framework to carry out any optimizations. Viton allows to focus on writing the actual code, without having to worry about data organization. Instead, it's sufficient to either infer the logic to write from a small snippet of the data or make use of Tuplex's exception resolution mechanisms [172, 46]. In the following, we show for our example query that extracts information about `ForkEvents` from the Github dataset that hyperspecialization and specifically the filter-promotion optimization introduced in §6.5.2 improves performance over a *globally* specialized data pipeline.

In the Github dataset, `PushEvents` un-suprisingly are the most frequent event type. However, the structure of

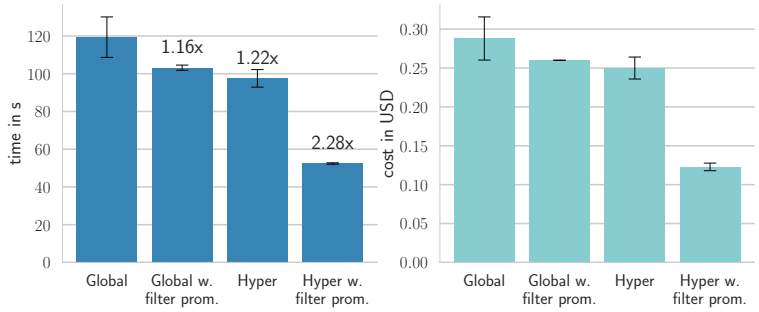


Figure 7.6: Github query showing the effect of using filter promotion using 24x parallelism on a specialization unit size of 2G.

`PushEvent`s changes drastically over time and is different from the one of `ForkEvent`s. This means, that Tuplex without hyperspecialization enabled wrongly deduces `PushEvent`s to be the most likely schema and optimizes its normal-case along `PushEvent`s but not the `ForkEvent`s that the query actually only cares about. As seen in figure 7.6, enabling filter-promotion globally allows the query to fit better to the relevant `ForkEvent`s. However, the nature of `ForkEvent`s changes over years. Interestingly, though fitting wrongly to `PushEvent`s, hyperspecialization itself is able to outperform a globally specialized pipeline by a factor of 1.22× with a slight edge over the filter-promotion. Combining hyper-specialization with filter-promotion allows to outperform the globally specialized pipeline significantly by a factor of more than 2× at comparable cost savings. In contrary to the query in §7.1 the complex semi-structured nature of `ForkEvent`s still requires a significant number of rows to be processed through the fallback path (the general-case path is specialized to PushEvents), which explains the factor of 2.28× compared to 3× for the flights scenario.

7.3 | Baseline comparison

To better understand Viton's performance relative to other Python-based frameworks that operate on a serverless setting, we chose to compare Viton against Lithops v2.9.0 [162, 158, 161, 190, 159], which is an open-source multi-cloud serverless framework in the spirit of a successor to PyWren [70] but is able to invoke Lambdas 3× faster and also has an improved result fetch mechanism compared to PyWren [160]. Other serverless engines like Lambada[114] or Starling [133] unfortunately are not available for experimentation and also do not support our workload. I.e., StarlingDB [133] is limited to executing TPC-H queries by generating C++ code for a limited set of queries that is then pre-compiled to be executed on a Python runtime. Lambada [114] uses Numba[83] to execute Python code more efficiently, but does support numbers only so far. Our flights query however requires extensive string processing.

We configure Lithops with identical settings to Viton (10GB of runtime memory on each Lambda, AWS Lambda as compute and S3 as storage backend) and invoke it in serverless mode from the same EC2 r5d.xlarge instance. For the baseline, we re-implemented the flights query from §7.1 using the primitives available in Lithops and in plain Python

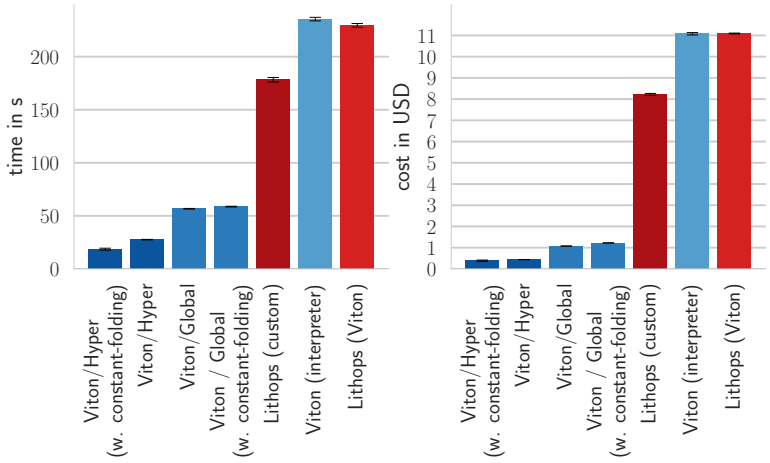


Figure 7.7: Viton can execute the same query 9.6× faster at less than 5% the cost of Lithops (PyWren).

with the utmost knowledge of the query and data. I.e., in our custom implementation the filter predicate is explicitly pushed-down to avoid costly S3 queries and processing data which is not required for any output row. In addition, we run Viton in fallback mode such that all rows are purely processed through the generated fallback code in Python. As Viton does not carry out any logical optimizations like operator reordering or pushdown in pure fallback mode, we port the generated code from Viton to Lithops as well to compare fairly. Figure 7.7 shows the results of the average of 10 runs in each setting, after performing an initial warmup run to avoid the cold-start of AWS Lambda containers. Viton in fallback mode performs a bit slower than Lithops due to initial sampling overheads, slower invocation of the fallback path and additional exception tracking logic which we didn't port over. Un-surprisingly writing better fitted logic to the concrete data improves cost and performance by 22% in Lithops (178s / $8.21 vs. 229s / $11.09).

With all optimizations enabled, despite the overhead of compiling code on each Lambda Viton outperforms end-to-end a native Python pipeline by a factor of 9.6× (18.6*s*). In addition, the use of efficient native code leads to a cost reduction by a factor of 21.6× at a cost of $0.38 on average.

7.4 | Performance breakdown

When we introduced hyperspecialization in §6.4, we discussed that implementing it successfully requires to carefully trade-off overheads. In table 7.1, we show a breakdown for a random flights query run with hyperspecialization on (and constant-folding). From §7.1 we recall that hyperspecialization leads indeed to a 3× time and cost reduction. We break down the overhead each Lambda has to perform into distinct categories. The "per λ" columns show the average across all 410 executed Lambda requests during a query. For this query, nearly half of the time is spent on sampling and specializing the code. We're certain that with additional engineering effort these efforts could be

	Total	Total $	per λ	$ per λ
Lambda execution time	2231.6s	$0.370	5.44s	$0.00091
deserialization	33.6s	$0.006	0.08s	$0.00001
sampling	495.9s	$0.083	1.21s	$0.00020
specialization	426.7s	$0.071	1.04s	$0.00017
LLVM (opt. + x86)	41.3s	$0.007	0.10s	$0.00002
Σ overheads	997.6s	$0.166	2.43s	$0.00041
fast path	1173.7s	$0.196	2.86s	$0.00048

Table 7.1: Breakdown of an example flights query execution which took 17.8s end-to-end with 410 parallel lambda executions.

lowered further, and see the overall numbers as encouraging.

7.5 | Revisiting sampling strategies

When comparing Viton to Tuplex, hyperspecialization mode in Viton is able to automatically detect a better sampling strategy by fitting better to the data. Revisiting the introductory experiment from figure 6.2, we compare in figure 7.8 the performance of Viton (using mode *C* on the Lambdas) configured with stratified sampling on the Lambdas (32MB sampling size, 10 samples per strata) and using Tuplex's 256KB sampling strategy on the client.

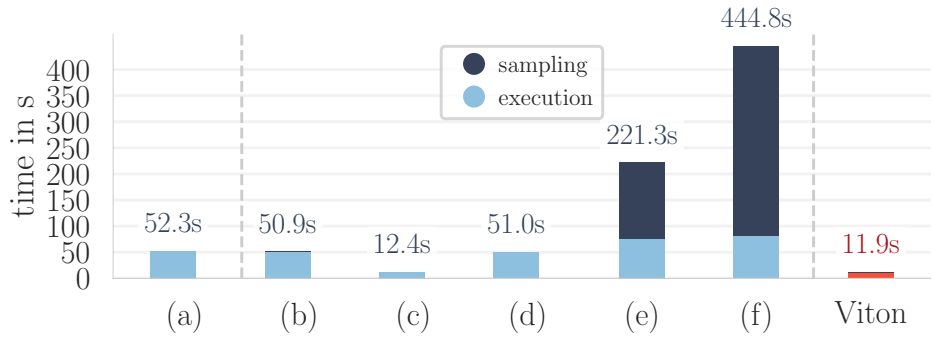


Figure 7.8: Depending on the sampling strategy users experience varying runtimes in Tuplex. Viton uses hyperspecialization to more robustly fit to marginal distributions across the input data.

In particular, we can explain the reason why Tuplex performs better in sampling mode *C* compared to *A*,*B* and *D* as more rows are placed on the slow fallback path, compared to mode *C* where all rows are processed on the compiled code paths (cf. table 7.3 and table 7.2). Running Viton with modes $A-D$ on the lambdas reduces variance of end-to-end time (we observe 11.3s – 14.8s on average for other modes) and with a similar configuration to (c) in fig. 7.8 Viton is even able to outperform Tuplex's best random pick slightly. Note that the runtime of Viton includes the

sampling mode	(a)	(b)	(c)	(d)	(e)	(f)
total time in s	52.34	50.93	12.35	51.00	221.32	444.76
thereof sampling in %	0.6%	1.0%	5.2%	2.5%	66.0%	81.9%
execution time in s	52.03	50.43	11.71	49.74	75.27	80.35
sampling time in s	0.31	0.50	0.64	1.27	146.05	364.41
fast path	87%	87%	98%	87%	87%	87%
slow path	2%	2%	2%	2%	2%	2%
fallback path	11%	11%	–	11%	11%	11%

Table 7.2: Path breakdown for different sampling strategies when using Tuplex.

sampling mode	(a)	(b)	(c)	(d)	(e)	(f)
fast path schema	C	B	A	B	B	B
slow path schema	D	D	E	D	D	D

schema	null	i64	f64	str	Option[i64]	Option[f64]	Option[str]
A	0	10	3	5	0	5	0
B	5	10	3	5	0	0	0
C	0	0	0	0	10	3	10
D	0	0	0	0	10	8	5

Table 7.3: Types detected for 1987 to 2021 for the flights query introduced in §7.1. `Option[T]` types in the schema cause more logic to be emitted and generally correspond to slower end-to-end runtimes.

time to perform sampling and hyperspecialization on each Lambda. By better fitting the code to subsets of the input data despite the specialization cost Viton pays on each Lambda, Viton is able to outperform Tuplex for this query.

7.6 | Discussion and Related work

In chapter 6 we showed that sampling-based Python UDF compilation faces challenges when it comes to processing in the cloud due to sample error and data heterogeneity. By fitting the code to the input data on each Lambda, hyperspecialization reduces overall execution time and monetary cost without any user effort. Our prototype, Viton, demonstrates that it is possible to compile efficient code on serverless functions that can help reduce costs and improve performance by a factor of 2–4×. In the following, we give an overview of other techniques that may be considered as well to deal with heterogeneity and why we think hyperspecialization is a valuable contribution to overcome limitations or provide an alternative for the large-scale data processing scenario.

Why not adaptive sampling? The DBMS community has proposed various approaches to *adapt* a query during runtime to changing data properties. Examples include micro-adaptivity in Vectorwise, where different precompiled flavors of primitives are kept that are chosen using a model at runtime upon each function invocation depending on statistics that are continuously updated [154]. Similarly, Grizzly generates and compiles on-the-fly different plan

versions and chooses then the most efficient version for stream processing based on hardware counters [52]. A more fine-grained approach than these two that avoids precompiled primitives and full at-runtime query (re)compilation is [104], where lightweight hooks and counters are used to permute already compiled queries efficiently, *e.g.*, by reordering code of individual predicates. Our approach in Viton, however, can't use precompiled primitives or an expensive adaptive scheme at runtime. In contrary to DBMS systems, Python UDFs often involve complex control flow, making it challenging or nearly impossible to utilize precompiled primitives. Furthermore, data-driven compilation requires a sample upfront in order to carry out compilation, in contrast to SQL queries that only require type information stored in a catalog. While it is often sufficient to update lightweight counters, like predicate selectivity or histograms to adapt SQL queries during query execution, adapting Python ASTs requires more complex counters for most AST nodes. Tracing JITs like PyPy [148] are suggested as an ideal design to achieve micro-adaptivity, yet in practice it was shown that their performance varies widely [172]. Tuplex's design therefore adopted a more coarse-grained sampling approach which allows to compile differently specialized code-paths ahead-of-time. Last, due to the complexity of UDF pipelines, their compilation times tend to be higher than for SQL queries. Thus, frequent (micro-)adaptation and compilation would likely make compile time could dominate overall query execution time.

Why not adaptive query processing? SkinnerDB [189, 188] proposes to use reinforcement learning to adapt on-the-fly join orders of a potentially suboptimal query plan. Spark 3.0 similarly introduces adaptive query execution to deal with data skew, balance buckets and switch out join strategies [35]. Morsel-driven parallelism allows to adapt parallelism on-the-fly [91]. These approaches are mostly concerned with either optimizing execution of higher-level operator graphs on-the-fly, or improve partitioning and scheduling during query execution. In a way these approaches are orthogonal and may benefit our system. In this work, we focus on improving the performance of individual stages by hyperspecializing to the input data on-the-fly which leads to more efficient code being executed.

Why serverless? Naturally, hyperspecialization as a technique can also be used on a single machine or cluster. However, serverless functions allow utilizing the elasticity of the cloud and the extra parallelism available to hit a sweet-spot in execution time vs. specialization opportunities. In future work, we're hoping to investigate how to split long-running tasks that cannot be specialized effectively, such as tasks that process files that internally have mixed schemas. This would entail a Lambda splitting its work and spawning additional Lambdas for different specializations. By combining such a splitting strategy with the higher efficiency of hyperspecialization, we believe that the potential of serverless compute can be pushed even further.

8 Conclusion

8.1 | Summary of contributions

This dissertation introduces a novel way of building efficient data analytics systems for Python. Properties of and heuristics about UDFs embedded within a higher-level data pipeline, enable to speculatively generate a normal-case code-path for the majority of rows together with a general and fallback path to process non-conforming rows. We call this approach data-driven compilation, as the key insight is to leverage both the query structure as well as statistics obtained from a small initial input data sample. We verified the viability of this approach through a prototype system, Tuplex, which has been open-sourced under `github.com/tuplex/tuplex`.

In the second part of this dissertation, we dove deeper into the question of how many code-paths are actually a good idea and how sensitive the data-driven compilation approach is to sampling errors. With Viton, we push the boundary further and enable data-driven compilation to work with heterogeneous datasets. We also extended the ideas of specialization to more complex semi-structured data to allow users to seamless work with JSON data by mapping them directly to Python dictionaries. Hyperspecialization shows that it is possible to overcome sampling-based Python UDF compilation challenges due to sample error and data heterogeneity. Viton uses AWS Lambda functions to distribute work. By fitting the code to the input data on each Lambda, hyperspecialization reduces overall execution time and monetary cost without any user effort.

Impact: Since Tuplex has been open-sourced it did gain traction within the open-source community [164, 100]. Moreover, our work has been used as a competitive benchmark in other research works like [39] and benchmarks developed within the original paper [172] have been used by other groups to design future data systems. The full paper has been reproduced by the TU Dresden database research group [197]. Particularly, the claim that Tuplex is one of the fastest data analytics frameworks for Python has been independently validated and Tuplex performed even better on their machine than on ours [197]. Research should be available and reproduced Tuplex was awarded all 3 ACM reproducibility badges (Artifacts Available, Results Reproduced, Artifacts Evaluated & Reusable) in 2022.

8.2 | Future research directions

Both Tuplex and Viton are just the start of a new, exicting resarch direction on building efficient data analytics systems using speculative compilation techniques. Indeed, there are many possibilities on what could be build on top of the existing infrastructure from this thesis:

Tuplex/Viton as execution engine: DataFrames have been widely embraced as an abstraction to process data. One of the most popular frameworks is Pandas [101], with the framework as popular as Python used to be in 2016 [121]. But Pandas is designed to be used single-threaded with a single Python interpreter process. For this reason, projects like Modin [134] or Magpie [68] partition data to distribute work and virtualize the API of Pandas such that alternative backends can be used to process Pandas operations. This allows to overcome limitations of the original Pandas API when it comes to both memory and compute. Because Tuplex is based on Python code, it can easily be used as a more competitive execution backend for such higher-level frameworks. Similarly, the ability to handle exceptions in a more robust way than existing frameworks make it potentially usable as execution layer for data cleaning and preperation services like Trifacta's wrangler [74], or for frameworks like Snorkel [145] which uses UDFs written in Python to generate labels.

Virtualization of 3rd party libraries with speculation: Snek [26] proposed the idea to virtualize external library calls and generate more efficient code through lightweight modular staging [150]. Currently, Tuplex does not support compilation of external libraries. However, many popular libraries like PyTorch, Tensorflow, Scipy, Numpy or XGBoost consist of primitives written in C that are surrounded by Python code to integrate them. By using the principle of data-driven compilation together with appropriate virtualization the glue code, the C-primitives could be desugared and inlined with the query Tuplex produces. Because of the tiered path design of Tuplex, it is not necessary to fully compile external modules, but rather it is sufficient to focus on the common-case.

Partitioning of data to fit hyperspecialization: In chapter 6, we introduced the idea of hyperspecialization and showed that efficient sampling strategies are a key issue to achieve sufficient performance. But, sampling information (and code paths) could be stored if data was pre-partitioned such that individual partitions are as homogenous as possible. Over the years multiple techniques like database cracking [59] propose to adapt data layout over time based by using queries as hints on how to physically reorganize data better [163]. Similarly, the information obtained from the normal-case and de-optimizations could be used to adaptively process incoming data and ensure data organization on ingestion to allow for future analytical runs to be performed more efficiently using the observed statistical properties of the dataset.

Extension of data-driven compilation for webservices and stream processing: In this dissertation the focus of the work has been primarily large-scale data analytics. Yet, many popular web frameworks like Django [38], Flask [51] or FastAPI [86] are also written in Python with the same performance issues. Famously, Dropbox decided to rewrite and replace critical pieces of their Python infrastructure with software written in Go [89] and Rust [61] for performance reasons related to Python. The ideas of specializing to the common-case however could also applied for example when serving get requests or within streaming systems when processing chunks of data.

Sample-based optimization: Within this dissertation multiple optimizations have been explored that are based on properties inferred from a sample of the data. Yet, there are many other optimizations that could be carried out like range-based integer optimizations, delayed parsing of fields, or loop-based speculations. Furthermore, in this dissertation we haven't explored yet any potential speculative vectorization using the data-driven compilation approach.

Cloud-based optimizer: Distributed computing is hard, and especially with the complex pay-per-query models offered by cloud-vendors there are many more dimensions along the questions of: How much memory to use? How to trade off cold-start, hot-starts versus longer periods of starting dedicated instances? When and what to compile of a query? With Viton demonstrating the viability of a compiled prototype of a serverless framework in Python for compiling to native code, an open research question is how to design an optimizer for for the cloud setting. I.e., an optimizer should help to plan where to compile/specialize which codepath to provide the best performance per $ around the typical pareto frontier of cloud-based engines [92]. This hinges on questions like whether we can find measures to decide whether to specialize on a dataset or not.

Bibliography

[1] Martín Abadi. "TensorFlow: learning functions at scale". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 2016, pp. 1–1.

[2] Yanif Ahmad and Christoph Koch. "DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases". In: *Proceedings of the VLDB Endowment* 2.2 (Aug. 2009), 1566–1569. ISSN: 2150-8097. DOI: 10.14778/1687553.1687592. URL: https://doi.org/10.14778/1687553.1687592.

[3] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. "NoDB: efficient query execution on raw data files". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 241–252.

[4] Kinan Dak Albab, Ishan Sharma, Justus Adam, Benjamin Kilimnik, Aaron Jeyaraj, Raj Paul, Artem Agvanian, Leonhard Spiegelberg, and Malte Schwarzkopf. "Pelton: Privacy-Compliant Storage For Web Applications By Construction". In: *to appear in OSDI'23*. USENIX. 2023.

[5] Anaconda, Inc. *Will Numba Work For My Code?* 2020. URL: http://numba.pydata.org/numba-doc/latest/user/5minguide.html#will-numba-work-for-my-code (visited on 05/16/2020).

[6] *Apache Arrow*. https://arrow.apache.org. 2022.

[7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 1383–1394.

[8] Nisha Arya. *Why you need to learn python in 2022*. 2022. URL: https://www.kdnuggets.com/2022/04/need-learn-python-2022.html.

[9] Ammar Askar. *Allow subtypes of unicode/str to hit the optimized unicode _ concatenate block*. https://bugs.python.org/issue27458. July 2016.

[10] Jean-Luc Aufranc. *Save the planet! Program in C, avoid Python, Perl*. https://www.cnx-software.com/2021/11/18/save-the-planet-program-in-c-avoid-python-perl/. 2021.

[11] Nikita Bajaj. *What makes python an ideal programming language for startups*. 2021. URL: https://www.kdnuggets.com/2021/12/makes-python-ideal-programming-language-startups.html.

[12] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. "Cython: The Best of Both Worlds". In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 31 –39. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.118.

[13] Max Bernstein. *How the Cinder JIT's function inliner helps us optimize Instagram*. https://engineering.fb.com/2022/05/02/open-source/cinder-jits-instagram/. May 2022.

[14] Jeffrey Werner Bezanson. "Abstraction in technical computing". PhD thesis. Massachusetts Institute of Technology, 2015.

[15] Poli Dey Bhavsar. *Why Python is One of the Most Preferred Languages for Data Science?* https://www.kdnuggets.com/2020/01/python-preferred-languages-data-science.html. Online; accessed 13 April 2023. Jan. 2020.

[16] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. "Tracing the meta-level: PyPy's tracing JIT compiler". In: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. 2009, pp. 18–25.

[17] Peter Boncz, Thomas Neumann, and Viktor Leis. "FSST: Fast Random Access String Compression". In: *Proc. VLDB Endow.* 13.12 (2020), 2649–2661. ISSN: 2150-8097. DOI: 10.14778/3407790.3407851. URL: https://doi.org/10.14778/3407790.3407851.

[18] Stefan Brunthaler. "Efficient Interpretation Using Quickening". In: *Proceedings of the 6th Symposium on Dynamic Languages*. DLS '10. Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, 1–14. ISBN: 9781450304054. DOI: 10.1145/1869631.1869633. URL: https://doi.org/10.1145/1869631.1869633.

[19] C. Chambers and D. Ungar. "Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language". In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. Portland, Oregon, USA, 1989, pp. 146–160. ISBN: 089791306X. DOI: 10.1145/73141.74831. URL: https://doi.org/10.1145/73141.74831.

[20] Lin Cheng, Berkin Ilbeyi, Carl Friedrich Bolz-Tereick, and Christopher Batten. "Type Freezing: Exploiting Attribute Type Monomorphism in Tracing JIT Compilers". In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. CGO 2020. San Diego, CA, USA: Association for Computing Machinery, 2020, 16–29. ISBN: 9781450370479. DOI: 10.1145/3368826.3377907. URL: https://doi.org/10.1145/3368826.3377907.

[21] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. "An Architecture for Compiling UDF-centric Workflows". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1466–1477.

[22] Databricks. *W3L1: Apache Logs Lab Dataframes (Python)*. 2019. URL: https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/2799933550853697/4438435960036599/2202577924924539/latest.html (visited on 03/24/2020).

[23] Databricks, Inc. *Handling bad records and files*. Jan. 2021. URL: https://docs.databricks.com/spark/latest/spark-sql/handling-bad-records.html (visited on 04/12/2021).

[24] Lorenzo De Stefani, Leonhard F Spiegelberg, Eli Upfal, and Tim Kraska. "VizCertify: A framework for secure visual data exploration". In: *2019 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE. 2019, pp. 241–251.

[25] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[26] James M Decker, Dan Moldovan, Andrew A Johnson, Guannan Wei, Vritant Bhardwaj, Gregory Essertel, and Fei Wang. *Snek: Overloading Python Semantics via Virtualization*. July 2019. URL: https://www.cs.purdue.edu/homes/rompf/papers/essertel-preprint201907.pdf.

[27] James M Decker, Dan Moldovan, Guannan Wei, Vritant Bhardwaj, Gregory Essertel, Fei Wang, Alexander B Wiltschko, and Tiark Rompf. "The 800 Pound Python in the Machine Learning Room". In: *ret* 10 (2018), p. 0.

[28] G. Dot, A. Martínez, and A. González. "Analysis and Optimization of Engines for Dynamically Typed Languages". In: *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Oct. 2015, pp. 41–48. DOI: 10.1109/SBAC-PAD.2015.20.

[29] Gem Dot, Alejandro Martínez, and Antonio González. "Analysis and Optimization of Engines for Dynamically Typed Languages". In: *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2015, pp. 41–48. DOI: 10.1109/SBAC-PAD.2015.20.

[30] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. "An intermediate representation for speculative optimizations in a dynamic compiler". In: *Proceedings of the 7th ACM workshop on Virtual Machines and Intermediate Languages*. 2013, pp. 1–10.

[31] Dominik Durner, Viktor Leis, and Thomas Neumann. "JSON Tiles: Fast Analytics on Semi-Structured Data". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21.

Virtual Event, China: Association for Computing Machinery, 2021, 445–458. ISBN: 9781450383431. DOI: 10.1145/3448016.3452809. URL: https://doi.org/10.1145/3448016.3452809.

[32] Christian Duta, Denis Hirn, and Torsten Grust. "Compiling pl/SQL away". In: *arXiv preprint arXiv:1909.03291* (2019).

[33] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. "Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 799–815.

[34] Julia Evans. *Pandas Cookbook: Chapter 7 – Cleaning up messy data*. Aug. 2020. URL: https://github.com/jvns/pandas-cookbook/blob/master/cookbook/Chapter%207%20-%20Cleaning%20up%20messy%20data.ipynb (visited on 03/24/2021).

[35] Wenchen Fan, Herman van Hoevell, and MaryAnn Xue. *Adaptive Query Execution: Speeding Up Spark SQL at Runtime*. 2020. URL: https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html.

[36] Tim Fischer, Denis Hirn, and Torsten Grust. "Snakes on a Plan: Compiling Python Functions into Plain SQL Queries". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, 2389–2392. ISBN: 9781450392495. DOI: 10.1145/3514221.3520175. URL: https://doi.org/10.1145/3514221.3520175.

[37] Daniela Florescu and Ghislain Fourny. "JSONiq: The history of a query language". In: *IEEE internet computing* 17.5 (2013), pp. 86–90.

[38] Jeff Forcier, Paul Bissex, and Wesley J Chun. *Python web development with Django*. Addison-Wesley Professional, 2008.

[39] Yannis Foufoulas, Alkis Simitsis, Lefteris Stamatogiannakis, and Yannis Ioannidis. "YeSQL: " you extend SQL" with rich and highly performant user-defined functions in relational databases". In: *Proceedings of the VLDB Endowment* 15.10 (2022), pp. 2270–2283.

[40] Python Software Foundation. *The Python Profilers: cProfile*. Apr. 2021. URL: https://docs.python.org/3/library/profile.html#module-cProfile (visited on 04/13/2021).

[41] Python Software Foundation. *Python/C API Reference Manual*. https://docs.python.org/3.9/c-api/index.html. 2023.

[42] Ghislain Fourny, David Dao, Can Berker Cikis, Ce Zhang, and Gustavo Alonso. "RumbleML: program the lakehouse with JSONiq". In: *arXiv preprint arXiv:2112.12638* (2021).

[43] Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. "Static Value Analysis of Python Programs by Abstract Interpretation". In: *NASA Formal Methods*. Ed. by Aaron Dutle, César Muñoz, and Anthony Narkawicz. Cham: Springer International Publishing, 2018, pp. 185–202. ISBN: 978-3-319-77935-5.

[44] Amir Gandomi and Murtaza Haider. "Beyond the hype: Big data concepts, methods, and analytics". In: *International Journal of Information Management* 35.2 (2015), pp. 137–144. ISSN: 0268-4012. DOI: https://doi.org/10.1016/j.ijinfomgt.2014.10.007. URL: https://www.sciencedirect.com/science/article/pii/S0268401214001066.

[45] Github. *Changelog - Github Docs*. June 2022. URL: https://docs.github.com/en/graphql/overview/changelog.

[46] Benjamin Givertz. "Incremental Exception Resolution in Tuplex". Brown University, 2022.

[47] Google. *google/pytype: A static type analyzer for Python code*. https://github.com/google/pytype. July 2016.

[48] Lars Grammel. *Wrangling US Flight Data - Part 1*. May 2015. URL: https://www.trifacta.com/blog/wrangling-us-flight-data-part-1/ (visited on 09/14/2019).

[49] W. Shane Grant and Randolph Voorhies. *cereal - A C++11 library for serialization*. http://uscilab.github.io/cereal/. 2017.

[50] Ilya Grigorik. *GH Archive*. https://www.gharchive.org/. 2023.

[51] Miguel Grinberg. *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.

[52] Philipp M Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. "Grizzly: Efficient stream processing through adaptive query compilation". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2487–2503.

[53] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362.

[54] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. "MaxSMT-Based Type Inference for Python 3". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 12–19. ISBN: 978-3-319-96142-2.

[55] Kay Hayen. *Nuitka*. 2018. URL: http://nuitka.net/ (visited on 05/12/2019).

[56] Anna Herlihy. *PYLLVM: A compiler from a subset of Python to LLVM-IR*. May 2016. URL: https://pycon.org.il/2016/static/sessions/anna-herlihy.pdf (visited on 05/12/2020).

[57] Silu Huang, Erkang (Eric) Zhu, Surajit Chaudhuri, and Leonhard Spiegelberg. *T-ReX: Optimizing Pattern Search on Time Series (Extended Version)*. Tech. rep. MSR-TR-2023-12. Microsoft, 2023. URL: https://www.microsoft.com/en-us/research/publication/t-rex-optimizing-pattern-search-on-time-series-extended-version/.

[58] Fabian Hueske, Mathias Peters, Matthias J Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. "Opening the black boxes in data flow optimization". In: *Proceedings of the VLDB Endowment* 5.11 (2012), pp. 1256–1267.

[59] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. "Database Cracking." In: *CIDR*. Vol. 7. 2007, pp. 68–78.

[60] Mohamed Ismail and G. Edward Suh. "Quantitative Overhead Analysis for Python". In: *IEEE International Symposium on Workload Characterization (IISWC)* (2018), pp. 36–47.

[61] Sujay Jayakar. *Rewriting the heart of our sync engine*. https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine. Mar. 2020.

[62] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, and Byung-Gon Chun. "JANUS: Fast and Flexible Deep Learning via Symbolic Graph Execution of Imperative Programs". In: *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*. 2019, pp. 453–468.

[63] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, Taebum Kim, and Byung-Gon Chun. "Speculative Symbolic Graph Execution of Imperative Deep Learning Programs". In: *SIGOPS Oper. Syst. Rev.* 53.1 (July 2019), pp. 26–33. ISSN: 0163-5980. DOI: 10.1145/3352020.3352025. URL: https://doi.org/10.1145/3352020.3352025.

[64] Mark Shannon Jeroen Demeyer and Petr Viktorin. *PEP 590: Vectorcall: a fast calling protocol for CPython*. https://docs.python.org/3/whatsnew/3.8.html#pep-590-vectorcall-a-fast-calling-protocol-for-cpython. Oct. 2019.

[65] Cy Jervis. *Introducing S3Guard: S3 Consistency for Apache Hadoop*. Aug. 2017. URL: https://blog.cloudera.com/introducing-s3guard-s3-consistency-for-apache-hadoop/ (visited on 05/26/2020).

[66] Jetbrains. *Python Developers Survey 2018 Results*. https://www.jetbrains.com/research/python-developers-survey-2018/. (Accessed on 07/24/2019). 2018.

[67] Li Jin. *Introducing Pandas UDF for PySpark - The Databricks Blog*. https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html. Oct. 2017. (Visited on 07/22/2019).

[68] Alekh Jindal, K Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, et al. "Magpie: Python at Speed and Scale using Cloud Backends." In: *CIDR*. 2021.

[69] Johannes Kepler University (JKU). *The Truffle Language Implementation Framework*. 2019. URL: http://www.ssw.uni-linz.ac.at/Research/Projects/JVM/Truffle.html (visited on 04/17/2019).

[70] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. "Occupy the cloud: Distributed computing for the 99%". In: *Proceedings of the 2017 Symposium on Cloud Computing*. ACM. 2017, pp. 445–451.

[71] Nicola Jones. "How to stop data centres from gobbling up the world's electricity". In: *Nature* 561.7722 (2018), pp. 163–167.

[72] Mayur P Joshi, Ning Su, Robert D Austin, and Anand K Sundaram. "Why so many data science projects fail to deliver". In: *MIT Sloan Management Review* 62.3 (2021).

[73] Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Soto Muoz, Victor Ng, Alex Ng, and Donna L Baker. *The definitive guide to Jython: Python for the Java platform*. Apress, 2010.

[74] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. "Wrangler: Interactive Visual Specification of Data Transformation Scripts". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Vancouver, BC, Canada: ACM, 2011, pp. 3363–3372. ISBN: 978-1-4503-0228-9. DOI: 10.1145/1978942.1979444. URL: http://doi.acm.org/10.1145/1978942.1979444.

[75] Sital Kedia, Shuojie Wang, and Avery Ching. *Apache Spark @Scale: A 60 TB+ production use case*. Aug. 2016. URL: https://code.fb.com/core-data/apache-spark-scale-a-60-tb-production-use-case/ (visited on 11/25/2018).

[76] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots". In: *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)*. 2011, pp. 195–206.

[77] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask". In: *Proceedings of the VLDB Endowment* 11.13 (2018), pp. 2209–2222.

[78] Steffen Kläbe, Bobby DeSantis, Stefan Hagedorn, and Kai-Uwe Sattler. "Accelerating Python UDFs in Vectorized Query Execution". In: (2022).

[79] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. "Building Efficient Query Engines in a High-Level Language". In: *Proceedings of the VLDB Endowment* 7.10 (June 2014), 853–864. ISSN: 2150-8097. DOI: 10.14778/2732951.2732959. URL: https://doi.org/10.14778/2732951.2732959.

[80] Sanjay Krishnan, Michael J. Franklin, Kenneth Goldberg, and Eugene Wu. *BoostClean: Automated Error Detection and Repair for Machine Learning*. Nov. 2017. arXiv: 1711.01299 [cs.DB].

[81] Mads R.B. Kristensen, Simon A.F. Lund, Troels Blum, and James Avery. "Fusion of Parallel Array Operations". In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. Haifa, Israel, 2016, pp. 71–85. ISBN: 9781450341219. DOI: 10.1145/2967938.2967945. URL: https://doi.org/10.1145/2967938.2967945.

[82] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A LLVM-based Python JIT Compiler". In: *Proceedings of the 2nd Workshop on the LLVM Compiler Infrastructure in HPC*. Austin, Texas, 2015, 7:1–7:6. ISBN: 978-1-4503-4005-2. DOI: 10.1145/2833157.2833162. URL: http://doi.acm.org/10.1145/2833157.2833162.

[83] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A LLVM-Based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. URL: https://doi.org/10.1145/2833157.2833162.

[84] Lukasz Langa. *What's New In Python 3.9*. https://docs.python.org/3/whatsnew/3.9.html#new-parser. Oct. 2020.

[85] Geoff Langdale and Daniel Lemire. "Parsing gigabytes of JSON per second". In: *The VLDB Journal* 28.6 (2019), pp. 941–960.

[86] Malhar Lathkar. "Getting Started with FastAPI". In: *High-Performance Web Apps with FastAPI: The Asynchronous Web Framework Based on Modern Python*. Springer, 2023, pp. 29–64.

[87] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. *MLIR: A Compiler Infrastructure for the End of Moore's Law*. 2020. arXiv: 2002.11054 [cs.PL].

[88] Chris Leary and Todd Wang. *XLA: TensorFlow, compiled*. 2017. URL: https://youtu.be/kAOanJczHA0 (visited on 07/15/2020).

[89] Patrick Lee. *Open Sourcing Our Go Libraries*. https://dropbox.tech/infrastructure/open-sourcing-our-go-libraries. July 2014.

[90] Jukka Lehtosalo, G v Rossum, Ivan Levkivskyi, Michael J Sullivan, David Fisher, Greg Price, Michael Lee, N Seyfer, R Barton, S Ilinskiy, et al. "Mypy-optional static typing for python". In: *URL: http://mypy-lang.org/[cited 2021-11-30]* (2017).

[91] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, 743–754. ISBN: 9781450323765. DOI: 10.1145/2588555.2610507. URL: https://doi.org/10.1145/2588555.2610507.

[92] Viktor Leis and Maximilian Kuschewski. "Towards cost-optimal query processing in the cloud". In: *Proceedings of the VLDB Endowment* 14.9 (2021), pp. 1606–1612.

[93] Neville Li. *Big Data Processing at Spotify: The Road to Scio (Part 1)*. https://engineering.atspotify.com/2017/10/big-data-processing-at-spotify-the-road-to-scio-part-1/. Oct. 2017.

[94] Tableau Software LLC. *Tableau Hyper API*. Apr. 2021. URL: https://help.tableau.com/current/api/hyper_api/en-us/index.html (visited on 04/07/2021).

[95] LLVM Project. *Itanium ABI Zero-cost Exception Handling*. Aug. 2019. URL: https://releases.llvm.org/8.0.1/docs/ExceptionHandling.html#itanium-abi-zero-cost-exception-handling (visited on 05/01/2020).

[96] LLVM Project. *Setjmp/Longjmp Exception Handling*. Aug. 2019. URL: https://releases.llvm.org/8.0.1/docs/ExceptionHandling.html#setjmp-longjmp-exception-handling (visited on 05/01/2020).

[97] Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. "Gradual soundness: Lessons from static python". In: *arXiv preprint arXiv:2206.13831* (2022).

[98] Louie Lu. *Internals of CPython*. https://hackmd.io/@klouielu/ByMHBMjFe?type=view. Mar. 2021.

[99] Federica Lucivero. "Big Data, Big Waste? A Reflection on the Environmental Sustainability of Big Data Initiatives". In: *Sci Eng Ethics 26, 1009–1030* (2020).

[100] Wes McKinney. *I've been keeping an eye out for this project (Tuplex, formerly TupleWare) since the first paper was published in 2014. Excited to see that it's finally been open sourced with an Apache 2.0 license:https://t.co/RchsLKIe6sNew paper with context here: https://t.co/lxnip0zPBL*. July 5, 2021. URL: https://twitter.com/wesmckinn/status/1412077521932374020. https://twitter.com/wesmckinn.

[101] Wes McKinney et al. "pandas: a foundational Python library for data analysis and statistics". In: *Python for high performance and scientific computing* 14.9 (2011), pp. 1–9.

[102] Wes McKinney et al. "pandas: a foundational Python library for data analysis and statistics". In: *Python for High Performance and Scientific Computing* 14.9 (2011).

[103] Erik Meijer, Brian Beckman, and Gavin Bierman. "LINQ: Reconciling Object, Relations and XML in the .NET Framework". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: Association for Computing Machinery, 2006, p. 706. ISBN: 1595934340. DOI: 10.1145/1142473.1142552. URL: https://doi.org/10.1145/1142473.1142552.

[104] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. "Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling". In: *Proc. VLDB Endow.* 14.2 (2020), 101–113. ISSN: 2150-8097. DOI: 10.14778/3425879.3425882. URL: https://doi.org/10.14778/3425879.3425882.

[105] Meta. *Cinder is Meta's internal performance-oriented production version of CPython.* https://github.com/facebookincubator/cinder. 2021.

[106] Micro Focus International, PLC. *Vertica 9.2.x: Capturing Load Rejections and Exceptions*. 2020. URL: https://www.vertica.com/docs/9.2.x/HTML/Content/Authoring/AdministratorsGuide/BulkLoadCOPY/CapturingLoadExceptionsAndRejections.htm (visited on 05/26/2020).

[107] Kevin Modzelewski. *Pyston 0.6.1 released, and future plans*. Jan. 2017. URL: https://blog.pyston.org/2017/01/31/pyston-0-6-1-released-and-future-plans/ (visited on 04/17/2019).

[108] Kevin Modzelewski. *Introducing Pyston: an upcoming, JIT-based Python implementation*. 2018. URL: https://blogs.dropbox.com/tech/2014/04/introducing-pyston-an-upcoming-jit-based-python-implementation/ (visited on 05/11/2020).

[109] Kevin Modzelewski. *Pyston v2: 20% faster Python*. Oct. 2020. URL: https://blog.pyston.org/2020/10/28/pyston-v2-20-faster-python/ (visited on 10/28/2020).

[110] Yann Moisan. *Spark performance tuning from the trenches*. May 2018. URL: https://medium.com/teads-engineering/spark-performance-tuning-from-the-trenches-7cbde521cf60 (visited on 03/24/2020).

[111] Philipp Moritz and Robert Nishihara. *Plasma: A High-Performance Shared-Memory Object Store*. https://arrow.apache.org/blog/2017/08/08/plasma-in-memory-object-store/. Aug. 2017.

[112] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. "Ray: A distributed framework for emerging {AI} applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 561–577.

[113] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. "Instant loading for main memory databases". In: *Proceedings of the VLDB Endowment* 6.14 (2013), pp. 1702–1713.

[114] Ingo Müller, Renato Marroquín, and Gustavo Alonso. "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, 115–130. ISBN: 9781450367356. DOI: 10.1145/3318464.3389758. URL: https://doi.org/10.1145/3318464.3389758.

[115] Thuwarakesh Murallie. *How to speed up python data pipelines up to 91X?* 2021. URL: https://towardsdatascience.com/how-to-speed-up-python-data-pipelines-up-to-91x-80d7accfe7ec.

[116] Myip.ms. *Blacklist IP Addresses Live Database (Real-time)*. 2020. URL: https://myip.ms/browse/blacklist (visited on 03/24/2020).

[117] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. "Gerenuk: thin computation over big native data using speculative program transformation". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 2019, pp. 538–553.

[118] Kyle Neath. *Product - Notifications & Stars*. https://github.blog/2012-08-06-notifications-stars/. Aug. 2012.

[119] Thomas Neumann. "Efficiently compiling efficient query plans for modern hardware". In: *Proceedings of the VLDB Endowment* 4.9 (2011), pp. 539–550.

[120] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. "Skyway: Connecting managed heaps in distributed big data systems". In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 56–69.

[121] Peter Olson. *Pandas Is Now As Popular As Python Was in 2016*. https://ponder.io/pandas-is-now-as-popular-as-python-was-in-2016. July 2022.

[122] NYC OpenData. *311 Service Requests from 2010 to Present*. 2020. URL: https://data.cityofnewyork.us/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9 (visited on 05/12/2020).

[123] Oracle. *GraalVM Implementation of Python*. https://github.com/oracle/graalpython. July 2018.

[124] Oracle, Inc. *GraalVM*. 2021. URL: https://www.graalvm.org/ (visited on 04/13/2021).

[125] Shoumik Palkar. *PR #308: Remove Multithreaded Backend*. Aug. 2018. URL: https://github.com/weld-project/weld/pull/383 (visited on 08/05/2020).

[126] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. "Evaluating End-to-end Optimization for Data Analytics Applications in Weld". In: *Proceedings of the VLDB Endowment* 11.9 (May 2018), pp. 1002–1015. ISSN: 2150-8097. DOI: 10.14778/3213880.3213890. URL: https://doi.org/10.14778/3213880.3213890.

[127] Shoumik Palkar and Matei Zaharia. "Optimizing Data-Intensive Computations in Existing Libraries with Split Annotations". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, 291–305. ISBN: 9781450368735. DOI: 10.1145/3341301.3359652. URL: https://doi.org/10.1145/3341301.3359652.

[128] Shoumik Palkar and Matei Zaharia. "Optimizing data-intensive computations in existing libraries with split annotations". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 2019, pp. 291–305.

[129] Arash Partow. *The Global Airport Database*. 2020. URL: https://www.partow.net/miscellaneous/airportdatabase/ (visited on 03/24/2020).

[130] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[131] Tejas Patil and Jing Zheng. *Using Apache Spark for large-scale language model training*. Feb. 2017. URL: https://code.fb.com/core-data/using-apache-spark-for-large-scale-language-model-training/ (visited on 11/25/2018).

[132] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. "Velox: meta's unified execution engine". In: *Proceedings of the VLDB Endowment* 15.12 (2022), pp. 3372–3384.

[133] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. "Starling: A Scalable Query Engine on Cloud Functions". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, 131–141. ISBN: 9781450367356. DOI: 10.1145/3318464.3380609. URL: https://doi.org/10.1145/3318464.3380609.

[134] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E Gonzalez, Joseph M Hellerstein, Anthony D Joseph, and Aditya Parameswaran. "Towards Scalable Dataframe Systems". In: *Proceedings of the VLDB Endowment* 13.11 ().

[135] Gregory Piatetsky. *Python leads the 11 Top Data Science, Machine Learning Platforms: Trends and analysis*. 2019. URL: https://www.kdnuggets.com/2019/05/poll-top-data-science-machine-learning-platforms.html.

[136] Gregory Piatetsky. *Python eats away at R: Top Software for Analytics, Data Science, Machine Learning in 2018: Trends and Analysis*. 2018. URL: https://www.kdnuggets.com/2018/05/poll-tools-analytics-data-science-machine-learning-results.html.

[137] picloud. *The Cloudpickle package*. URL: https://github.com/cloudpipe/cloudpickle (visited on 11/25/2018).

[138] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. "Python: The full monty". In: *ACM SIGPLAN Notices* 48.10 (2013), pp. 217–232.

[139] PostgreSQL Global Development Group. *Error logging in COPY*. May 2015. URL: https://wiki.postgresql.org/wiki/Error_logging_in_COPY (visited on 05/26/2020).

[140] Elvis Pranskevichus. *What's New In Python 3.7*. https://docs.python.org/3/whatsnew/3.7.html#optimizations. June 2018.

[141] Elvis Pranskevichus and Yury Selivanov. *What's New In Python 3.6*. https://docs.python.org/3/whatsnew/3.6.html. Dec. 2016.

[142] Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. "Tracing Dynamic Features in Python Programs". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, 292–295. ISBN: 9781450328630. DOI: 10.1145/2597073.2597103. URL: https://doi-org.revproxy.brown.edu/10.1145/2597073.2597103.

[143] Karthik Ramachandra and Kwanghyun Park. "Blackmagic: Automatic inlining of scalar udfs into sql queries with froid". In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 1810–1813.

[144] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. "Froid: Optimization of imperative programs in a relational database". In: *Proceedings of the VLDB Endowment* 11.4 (2017), pp. 432–444.

[145] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. "Snorkel: Rapid training data creation with weak supervision". In: *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*. Vol. 11. 3. NIH Public Access. 2017, p. 269.

[146] Jose Manuel Redondo and Francisco Ortin. "A Comprehensive Evaluation of Common Python Implementations". In: *IEEE Software* 32.4 (2015), pp. 76–84. DOI: 10.1109/MS.2014.104.

[147] Kristian FD Rietveld and Harry AG Wijshoff. "Redefining The Query Optimization Process". In: *arXiv preprint arXiv:2203.01079* (2022).

[148] Armin Rigo, Maciej Fijalkowski, Carl Friedrich Bolz, Antonio Cuni, Benjamin Peterson, Alex Gaynor, Hakan Ardoe, Holger Krekel, and Samuele Pedroni. *PyPy*. 2021. URL: http://pypy.org/ (visited on 04/13/2021).

[149] Matthew Rocklin. "Dask: Parallel computation with blocked algorithms and task scheduling". In: *Proceedings of the 14th python in science conference*. Vol. 130. Citeseer. 2015, p. 136.

[150] Tiark Rompf and Martin Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs". In: *Communications of the ACM* 55.6 (2012), pp. 121–130.

[151] Nate Rosidi. *Data Science programming languages and when to use them*. 2022. URL: https://www.kdnuggets.com/2022/02/data-science-programming-languages.html.

[152] Lukasz Langa Guido van Rossum Jukka Lehtosalo. *PEP 484 – Type Hints*. https://peps.python.org/pep-0484/.

[153] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. *Glow: Graph lowering compiler techniques for neural networks*. 2018. arXiv: 1805.00907 [cs.PL].

[154] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. "Micro Adaptivity in Vectorwise". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: Association for Computing Machinery, 2013, 1231–1242. ISBN: 9781450320375. DOI: 10.1145/2463676.2465292. URL: https://doi.org/10.1145/2463676.2465292.

[155] Pablo Galindo Salgado. *What's New In Python 3.10¶*. https://docs.python.org/3/whatsnew/3.10.html#optimizations. Oct. 2021.

[156] Pablo Galindo Salgado. *What's New In Python 3.11*. https://docs.python.org/3/whatsnew/3.11.html. Apr. 2023.

[157] Michael Salib. "Starkiller: A static type inferencer and compiler for Python". PhD thesis. Massachusetts Institute of Technology, 2004.

[158] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Roca-Llaberia, and A. Arjona. "Toward Multicloud Access Transparency in Serverless Computing". In: *IEEE Software* 38.01 (2021), pp. 68–74. ISSN: 1937-4194. DOI: 10.1109/MS.2020.3029994.

[159] Josep Sampe. *Lithops*. https://github.com/lithops-cloud/lithops. Oct. 2018.

[160] Josep Sampe, Marc Sanchez-Artigas, Gil Vernik, Ido Yehekzel, and Pedro Garcia-Lopez. "Outsourcing Data Processing Jobs with Lithops". In: *IEEE Transactions on Cloud Computing* (2021), pp. 1–1. DOI: 10.1109/TCC.2021.3129000.

[161] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. "Serverless Data Analytics in the IBM Cloud". In: *Proceedings of the 19th International Middleware Conference Industry*. Middleware '18. Rennes, France: Association for Computing Machinery, 2018, 1–8. ISBN: 9781450360166. DOI: 10.1145/3284028.3284029. URL: https://doi.org/10.1145/3284028.3284029.

[162] Josep Sampé, Marc Sánchez-Artigas, Gil Vernik, Ido Yehekzel, and Pedro García-López. "Outsourcing Data Processing Jobs With Lithops". In: *IEEE Transactions on Cloud Computing* 11.1 (2023), pp. 1026–1037. DOI: 10.1109/TCC.2021.3129000.

[163] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. "The uncracked pieces in database cracking". In: *Proceedings of the VLDB Endowment* 7.2 (2013), pp. 97–108.

[164] Malte Schwarzkopf. *Today, we release Tuplex (https://t.co/fJgqR2KKCp), a new system that turns Python data science pipelines into efficient, optimized machine code.Think PySpark or Dask, but with end-to-end LLVM code generation for custom Python code. Tuplex often yields speedups of 10-20x. ?? pic.twitter.com/Tm65MgNfbI*. July 1, 2021. URL: https://twitter.com/ms705/status/1410658219324751880. https://twitter.com/ms705.

[165] ScrapeHero. *How to Scrape Real Estate Listings from Zillow.com using Python and LXML*. Oct. 2017. URL: https://www.scrapehero.com/how-to-scrape-real-estate-listings-on-zillow-com-using-python-and-lxml/ (visited on 09/12/2018).

[166] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. "How to Architect a Query Compiler". In: *Proceedings of the 2016 International Conference on Management of Data*. San Francisco, California, USA, 2016, pp. 1907–1922. ISBN: 9781450335317. DOI: 10.1145/2882903.2915244. URL: https://doi.org/10.1145/2882903.2915244.

[167] Mark Shannon. *Implementation plan for speeding up CPython*. https://github.com/markshannon/faster-cpython/blob/master/plan.md. Oct. 2020.

[168] Mark Shannong. *PEP 659 – Specializing Adaptive Interpreter*. https://peps.python.org/pep-0659/. Apr. 2021.

[169] Yunzhi Shao. "Speculative Compilation of Complex UDFs in Python Data Science". Brown University, 2022.

[170] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. "Vectorization vs. Compilation in Query Execution". In: *Proceedings of the 7th International Workshop on Data Management on New Hardware*. New York, NY, USA: Association for Computing Machinery, 2011, 33–40. ISBN: 9781450306584. DOI: 10.1145/1995441.1995446. URL: https://doi.org/10.1145/1995441.1995446.

[171] Leonhard Spiegelberg, Tim Kraska, and Malte Schwarzkopf. "Viton: A hyper-specializing execution engine for the serverless age". In: *under submission*. 2023.

[172] Leonhard Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. "Tuplex: Data Science in Python at Native Code Speed". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. Virtual Event, China: Association for Computing Machinery, 2021, 1718–1731. ISBN: 9781450383431. DOI: 10.1145/3448016.3457244. URL: https://doi.org/10.1145/3448016.3457244.

[173] Leonhard F Spiegelberg and Tim Kraska. "Tuplex: robust, efficient analytics when Python rules". In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 1958–1961.

[174] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. "Partial escape analysis and scalar replacement for Java". In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 2014, pp. 165–174.

[175] Victor Stinner. *CPython 3.10 repository*. https://github.com/python/cpython/blob/3.10/Python/ceval.c#L2072. Oct. 2010.

[176] Victor Stinner. *ceval.c: implement fast path for integers with a single digit*. https://bugs.python.org/issue21955. July 2014.

[177] Konstantinos Stylianou, Leonhard Spiegelberg, Maurice Herlihy, and Nic Carter. "Cryptocurrency competition and market concentration in the presence of network effects". In: *Ledger* 6 (2021), pp. 81–101.

[178] Martin Franz Sulzmann. *A general framework for Hindley/Milner type systems with constraints*. Yale University, 2000.

[179] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. "How to Architect a Query Compiler, Revisited". In: *Proceedings of the 2018 International Conference on Management of Data*. Houston, TX, USA, 2018, 307–322. ISBN: 9781450347037. DOI: 10.1145/3183713.3196893. URL: https://doi.org/10.1145/3183713.3196893.

[180] The PyPy Team. *PyPy: Performance*. 2021. URL: https://www.pypy.org/performance.html (visited on 04/13/2021).

[181] *The Python Language Reference - 3. data model*. 2022. URL: https://docs.python.org/3.9/reference/datamodel.html.

[182] *threading — Thread-based parallelism*. 2022. URL: https://docs.python.org/3/library/threading.html.

[183] Ehsan Totoni. *Bodo: Automatic HPC Performance and Scaling for Data Processing in Python (Ehsan Totoni)*. https://www.youtube.com/watch?v=DJ1sGQryoAc&list=PLSE8ODhjZXjbDOFN4U4-Uv95-N8sgzs5D&index=5. Oct. 2021.

[184] United States Department of Transportation Bureau of Transportation Statistics. *Carrier history lookup table*. 2020. URL: https://www.transtats.bts.gov/Download_Lookup.asp?Lookup=L_CARRIER_HISTORY (visited on 03/24/2020).

[185] United States Department of Transportation Bureau of Transportation Statistics. *Reporting Carrier On-Time Performance (1987-present)*. 2020. URL: https://www.transtats.bts.gov/Fields.asp?Table_ID=236 (visited on 03/24/2020).

[186] United States Department of Transportation Bureau of Transportation Statistics. *Understanding the Reporting of Causes of Flight Delays and Cancellations*. Jan. 2022. URL: https://www.bts.gov/topics/airlines-and-airports/understanding-reporting-causes-flight-delays-and-cancellations (visited on 03/01/2022).

[187] trstovall. *[bug] dask.dataframe.DataFrame.merge fails for inner join*. Mar. 2019. URL: https://github.com/dask/dask/issues/4643 (visited on 05/12/2020).

[188] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. "SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, 1153–1170. ISBN: 9781450356435. DOI: 10.1145/3299869.3300088. URL: https://doi.org/10.1145/3299869.3300088.

[189] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. "SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning". In: *ACM Trans. Database Syst.* 46.3 (2021). ISSN: 0362-5915. DOI: 10.1145/3464389. URL: https://doi.org/10.1145/3464389.

[190] Gil Vernik. *Process large data sets at massive scale with PyWren over IBM Cloud Functions*. Apr. 2018. URL: https://www.ibm.com/cloud/blog/process-large-data-sets-massive-scale-pywren-ibm-cloud-functions (visited on 03/01/2022).

[191] Gil Vernik, Michael Factor, Elliot K. Kolodner, Effi Ofer, Pietro Michiardi, and Francesco Pace. "Stocator: An Object Store Aware Connector for Apache Spark". In: *Proceedings of the 2017 Symposium on Cloud Computing*. Santa Clara, California, 2017, p. 653. ISBN: 9781450350280. DOI: 10.1145/3127479.3134761. URL: https://doi.org/10.1145/3127479.3134761.

[192] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. "Design and Evaluation of Gradual Typing for Python". In: *SIGPLAN Not.* 50.2 (2014), 45–56. ISSN: 0362-1340. DOI: 10.1145/2775052.2661101. URL: https://doi.org/10.1145/2775052.2661101.

[193] Jiannan Wang, Sanjay Krishnan, Michael J Franklin, Ken Goldberg, Tim Kraska, and Tova Milo. "A sample-and-clean framework for fast and accurate query processing on dirty data". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 469–480.

[194] Yin Wang. *PySonar2 - a semantic indexer for Python with interprocedural type inference*. https://github.com/yinwang0/pysonar2. Oct. 2013.

[195] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. "WeTune: Automatic Discovery and Verification of Query Rewrite Rules". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, 94–107. ISBN: 9781450392495. DOI: 10.1145/3514221.3526125. URL: https://doi.org/10.1145/3514221.3526125.

[196] David Wilson. *csvmonkey - Header-only vectorized, lazy-decoding, zero-copy CSV file parser*. May 2019. URL: https://github.com/dw/csvmonkey.

[197] Lucas Woltmann, Axel Hertsschuch, and Wolfgang Lehner. *Reproducibility Report for ACM SIGMOD 2021 Paper: "Tuplex: Data Science in Python at Native Code Speed"*. https://reproducibility.sigmod.org/rep_rep/2022/Lehner-SIGMODReproReport16.pdf. 2022.

[198] Eugene Wu, Samuel Madden, and Michael Stonebraker. "A demonstration of DBWipes: clean as you query". In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 1894–1897.

[199] Dong Xie, Badrish Chandramouli, Yinan Li, and Donald Kossmann. "Fishstore: Faster ingestion with subset hashing". In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1711–1728.

[200] Reynold Xin. *Project Zen: Making Spark Pythonic*. https://www.youtube.com/watch?v=-vJLTEOdLvA. Nov. 2021.

[201] Reynold Xin and Josh Rosen. *Project Tungsten: Bringing Apache Spark Closer to Bare Metal*. Apr. 2015. URL: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html (visited on 08/07/2019).

[202] Yi Yang, Ana Milanova, and Martin Hirzel. "Complex Python Features in the Wild". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. MSR '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, 282–293. ISBN: 9781450393034. DOI: 10.1145/3524842.3528467. URL: https://doi.org/10.1145/3524842.3528467.

[203] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language". In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, California, USA, Dec. 2008.

[204] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2012.

[205] Qiang Zhang, Lei Xu, Xiangyu Zhang, and Baowen Xu. "Quantifying the interpretation overhead of Python". In: *Science of Computer Programming* 215 (2022), p. 102759. URL: https://www.sciencedirect.com/science/article/pii/S0167642321001520.

[206] M. Zukowski, M. van de Wiel, and P. Boncz. "Vectorwise: A Vectorized Analytical DBMS". In: *Proceedings of the 28th IEEE International Conference on Data Engineering*. 2012, pp. 1349–1350.