Tuplex: Data Science in Python at Native Code Speed

Leonhard Spiegelberg
lspiegel@cs.brown.edu
Brown University

Rahul Yesantharao
rahuly@mit.edu
MIT

Malte Schwarzkopf
malte@cs.brown.edu
Brown University

Tim Kraska
kraska@mit.edu
MIT

Abstract

Data science pipelines today are either written in Python or contain user-defined functions (UDFs) written in Python. But executing interpreted Python code is slow, and arbitrary Python UDFs cannot be compiled to optimized machine code.

We present Tuplex, the first data analytics framework that just-in-time compiles developers' natural Python UDFs into efficient, end-to-end optimized native code. Tuplex introduces a novel *dual-mode execution model* that compiles an optimized fast path for the common case, and falls back on slower exception code paths for data that fail to match the fast path's assumptions. Dual-mode execution is crucial to making end-to-end optimizing compilation tractable: by focusing on the common case, Tuplex keeps the code simple enough to apply aggressive optimizations. Thanks to dual-mode execution, Tuplex pipelines always complete even if exceptions occur, and Tuplex's post-facto exception handling simplifies debugging.

We evaluate Tuplex with data science pipelines over real-world datasets. Compared to Spark and Dask, Tuplex improves end-to-end pipeline runtime by 5×–109×, and comes within 22% of a hand-optimized C++ baseline. Optimizations enabled by dual-mode processing improve runtime by up to 3×; Tuplex outperforms other Python compilers by 5×; and Tuplex performs well in a distributed setting.

1 Introduction

Data scientists today predominantly write code in Python, as the language is easy to learn and convenient to use. 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. This infrastructure imposes a heavy overhead, particularly if Python *user-defined functions* (UDFs) are inlined in a larger parallel computation, such as a Spark [70] job. For example, a PySpark job over flight data [63] might compute a flight's distance covered from kilometers to miles via a UDF after joining with a carrier table:

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

This code will run data loading and the join using Spark's compiled Scala operators, but must execute the Python UDF passed to `withColumn` in a Python interpreter. This requires passing data between the Python interpreter and the JVM [41], and prevents generating end-to-end optimized code across the UDFs—for example, an optimized pipeline might apply the UDF to `distance` while loading data from `flights.csv`.

Could we instead generate native code 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 that handles all possible code paths through a Python program is not tractable because of the complexity of Python's dynamic typing. 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 Python interpreter handles these possibilities through extra checks and exception handlers, but the sheer number of cases to handle makes it difficult to compile optimized code even for this simple UDF.

Tuplex is a new analytics framework that generates optimized *end-to-end* native code for data analytics pipelines with Python UDFs. Developers write their Tuplex pipelines using a LINQ-style API similar to PySpark's, and use Python UDFs without any type annotations. Tuplex compiles these pipelines into efficient native code by relying on 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. Our key insight is that separating these cases and leveraging the regular structure of LINQ-style data analytics pipelines makes Python UDF compilation tractable, as the Tuplex compiler faces a simpler and more constrained problem than a general Python compiler.

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. Second, Tuplex's generated native code must represent a semantically-correct Python execution in the interpreter. To guarantee this, Tuplex separates the input data into records for which the native code's behavior is identical to Python's, and ones for which it is not 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 records for resolution. Fourth, Tuplex must generate code with high optimization potential but also achieve fast JIT compilation, which it does using tuned LLVM compilation.

In addition to enabling compilation, dual-mode processing 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.

While the focus of this paper is primarily on multi-threaded processing efficiency of a single server, Tuplex is a distributed system, and we show results for a preliminary backend based on AWS lambda functions.

In summary, 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.
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.

We evaluated our Tuplex prototype over real-world datasets, including Zillow real estate adverts, a decade of U.S. flight data [63], and web server logs from a large university. Tuplex outperforms single-threaded Python and Pandas by 6.5–20.6×, and parallel Spark and Dask by 9.4–109× (§6.1). Tuplex outperforms general-purpose Python compilers by 5–32×, and its generated code comes within 2× of the performance of Weld [49] and Hyper [26] for pure query exection time, while achieving 2× faster end-to-end runtime in a realistic data analytics setting (§6.2). Tuplex's dual-mode processing facilitates end-to-end optimizations that improve runtime by up to 3× over simple UDF compilation (§6.3). Finally, Tuplex performs well on a single server and distributedly across a cluster of AWS Lambda functions (§6.4); and anecdotal evidence suggests that it simplifies the development and debugging of data science pipelines (§7). We will release Tuplex as open-source software.

2 Background and Related Work

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 1 summarizes the key points.

Python compilers. Building compilers for arbitrary Python programs, which lack the static types required for optimizing compilation, is challenging. PyPy [54] 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 [16] uses the Truffle [61] language interpreter to implement a similar approach while generating JVM bytecode for JIT compilation. Numba [31] JIT-compiles Python bytecode for annotated functions on which it can perform type inference. Numba supports a subset of Python and targets array-structured data from numeric libraries like NumPy [2]. 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). While Pyston [40] sought to create a full Python compiler using LLVM, it was abandoned due to insufficient performance, memory management problems, and complexity [39].

Python transpilers. Other approaches seek to cross-compile Python into languages for which optimizing compilers exist. Cython [4] unrolls the CPython interpreter and a Python module into C code, which interfaces with standard Python code. Nuitka [18] 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 [24], but such precompiled code precludes end-to-end optimization. Data-parallel intermediate representations (IRs) such as Weld [49] and MLIR [32] seek to address this problem. Weld, for example, allows cross-library optimization and generates code

System Class	Examples	Limitations
Tracing JIT Compiler	PyPy [54], Pyston [40]	Requires tracing to detect hotspots, cannot reason about high-level program structure, generated code must cover full Python semantics (slow).
Special Purpose JIT Compiler	Numba [31], XLA [33], Glow [55]	Only compiles well-formed, statically typed code, enters interpreter otherwise; use their own semantics, which often deviate from Python's.
Python Transpiler	Cython [4], Nuitka [18]	Unrolled interpreter code is slow and uses expensive Python object representation.
Data-parallel IR	Weld [49], MLIR [32]	No compilation from Python; static typing and lack exception support.
SQL Query Compiler	Flare [12], Hyper [45]	No Python UDF support.
Simple UDF Compiler	Tupleware [6]	Only supports UDFs for which types can be inferred statically, only numerical types, no exception support, no polymorphic types (*e.g.*, NULL values).

Table 1: Classes of system that compile 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.

that targets a common runtime and data representation, but requires libraries to be rewritten in Weld IR. Rather than requiring library rewrites, Mozart [50] 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. Many query compilers from SQL queries to native code exist [1, 28, 57, 59, 71], and some integrate with frameworks like Spark [12]. The primary concern of these compilers is to iterate efficiently over preorganized data [27, 58], and all lack UDF support, or merely provide interfaces to call precompiled UDFs written 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 optimiziation exists [20], these strategies fail on Python code. Tupleware [6] 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 [19], and it neither supports common cases like optional (NULL-valued) inputs, nor strings, nor can it handle exceptions in UDFs.

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 [8, 38, 53] 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 NULL values). Graal speculatively optimizes for exceptions [11] via polymorphic inline caches—an idea also used in the V8 Javascript engine—but the required checks and guards impose around a 30% overhead [10]. Finally, various dedicated systems track the impact of errors on models [29] or provide techniques to compute queries over dirty data [66, 67], but they do not integrate with compiled code.

Speculative processing. Programming language research on speculative compilation pioneered native code performance for dynamically-typed languages. Early approaches, like SELF [5], 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 particular "hot" parts of the code only.

A simpler approach than trying to compile general Python is to have Python merely act as a frontend calling a more efficient backend. However, to account for Python's dynamic types, the system has to speculate on how to call into this backend. Janus [21, 22] applies this idea to TensorFlow, and Snek [9] 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. However, 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. In addition, the backend's APIs may impose materialization points, which can reduce performance as they enforce unnecessary 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 [50]. Gerenuk [44] and Skyway [46] 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 records not suitable for this fast path to the interpreter. This simplifies the task sufficiently to make optimizing compilation tractable.

Tuplex supports natural Python code, rather than just 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 in its execution model, and handles them without compromising the performance of compiled code, as it collects records affected by exceptions and batches the slow path, rather than entering the interpreter from the fast path.

3 Tuplex Overview

Tuplex is a data analytics framework with a similar user experience to *e.g.*, PySpark, Dask, or DryadLINQ [69]. 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`). For example, the PySpark pipeline shown in §1 corresponds to the following Tuplex code:

```
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')
```

Like other systems, Tuplex partitions the input data (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 make this possible, Tuplex relies on a **dual-mode processing** model structured around two distinct code paths:

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

To establish what constitutes the normal case, Tuplex samples the input data and, based on the sample, determines the expected types and control flow on the normal-case code path. 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 code path. It lowers both to optimized machine code via the LLVM compiler framework.

Tuplex then executes the pipeline. The generated classifier code performs a single, cheap initial check on each row to determine if it can proceed on the normal-case path. Any rows that fail this check are placed in an exception pool for later processing, while the majority of rows proceed on the optimized normal-case path. If any exceptions occur on the normal-case code path, 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.

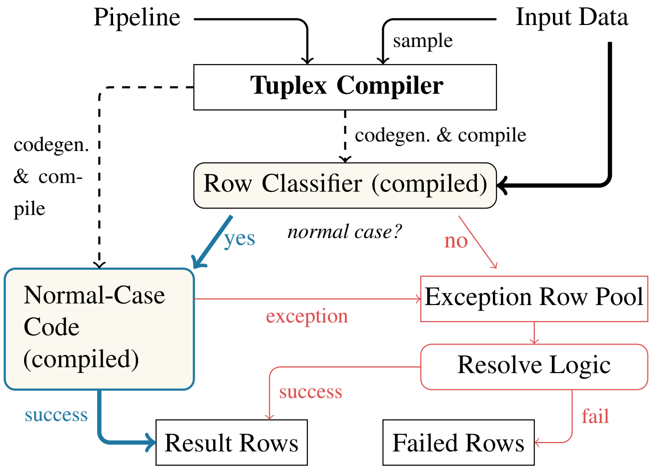


Figure 1: Tuplex uses an input sample to compile specialized code for the normal-case path (blue, left), which processes most rows, while the exception-case path (red, right) handles the remaining rows. Compiled parts are shaded in yellow.

In our example UDF, a malformed flight record 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 `NULL`, enters the normal-case path if the sample contained a mix of non-`NULL` and `NULL` values. However, the normal-case code path 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 an optional resolver:

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

Tuplex then merges the resolved rows into the results. If no resolver is provided, Tuplex reports the failed rows separately.

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 code path to common-case assumptions simplifies the generated logic by deferring complexity to the exception-case path, which makes JIT compilation tractable and allows for aggressive optimization.

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[1].

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

4.2 Establishing the Normal Case

The most important guidance for Tuplex to decide what code to generate for the normal-case path 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.

Row Structure. Input data may be dirty and contain different column counts. Tuplex counts the columns in each sample row and builds a histogram; it then picks the most common 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 uses a set of heuristics. For example, numeric strings containing periods are floats, zero/one integers and true/false are booleans, strings containing JSON are dictionaries, and empty values or explicit `NULL` strings are null 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, and because the actual type in the input data may be different from the developer's assumptions. In our example, 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 (`NULL`). Optional column values ("nullable" columns) 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 the necessary checks.

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.
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 generated 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 the exception path, and Tuplex uses it to more efficiently resolve some exception rows.

Row Classifier. All input rows must be classified 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

[1] These assumptions do not preclude aggregations, as discussed in §4.6.

adds complexity and disrupts control flow.

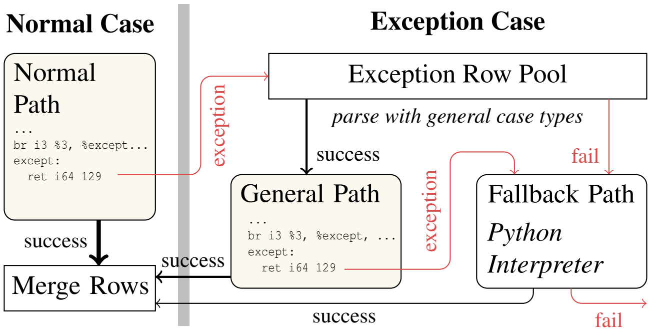
Code Paths. All of Tuplex's generated code must obey the top-level invariant that execution must match Python semantics. Tuplex traverses the Python AST for each UDF and generates matching LLVM IR for the language constructs it encounters. Where types are required, Tuplex instantiates them using the 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 code 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 code path also contains code for any user-provided resolvers whose implementation is a compilable UDF. However, the compiled general-case code cannot handle all exceptions, and must defer rows from the exception pool that it cannot process. The general-case code path includes logic to detect these cases, convert the data to Python object format, and invoke the Python interpreter inline.

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 meet the invariant of simulating a Python interpreter execution, the code Tuplex generates and executes for a row must have no observable effects that are distinct 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 2 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 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.



```
...
br i3 %3, %except...
except:
  ret i64 129
```

```
...
br i3 %3, %except, ...
except:
  ret i64 129
```

Figure 2: 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.

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 Execution

Executing a pipeline in Tuplex involves typical steps for a data analytics framework, though 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 common-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 §4.5).

To delineate stages, Tuplex follows a model similar to HyPer's [45]. Tuplex makes stages as long as possible, so that a row is processed through many UDFs while in the CPU cache, and to facilitate compiler optimizations across UDFs. In the ideal case, the bulk of input rows proceeds through a single, highly-optimized stage that ends with the materialized output of the pipeline.

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 (in terms of input rows) as the build side and terminates a stage at the materialized join input.

This standard design, however, requires adaptation to work with 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 the dual-mode execution model, 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 $NC(A_i)$.

4.6 Aggregates

Dual-mode processing works for aggregations as long as the aggregation function is associative. Tuplex can separately aggregate normal-case rows and, subsequently, exception-case rows via the general and fallback code paths; in a final merge step, it can combine the partial aggregates into a final result. This merging of partial aggregates can happen at the end of the stage (which requires immediate resolution of exception rows), or can be pushed further down the pipeline.

Aggregations are also compatible with Tuplex’s assumption that UDFs are stateless, as the framework can track 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 is responsible for managing the memory of `acc`, and the UDF remains stateless.

4.7 Optimizations

Tuplex applies both logical and compiler optimizations, particularly 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. In another, more complex optimization, Tuplex pushes UDFs that modify a column past any operators and UDFs that do not read it. This allows *e.g.*, pushing UDFs that rewrite non-key columns below joins, which is a good choice if the join is selective.[2] Crucially, this optimization is only possible because Tuplex analyzes the Python UDF code.

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 instruction 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 consequently avoids these optimizations.

5 Implementation

We implemented a prototype of Tuplex in about 65,000 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 like a regular module in Python or from a Jupyter Notebook. In addition, Tuplex provides a shell in CPython interactive mode. The prototype also offers a web UI and a history server, which developers can use to inspect their pipelines’ execution and any failed rows generated.

We built our prototype as a standalone data analytics system rather than integrating with an existing system like Spark or Dask because adding dual-mode processing to these systems would have required substantial code changes.

[2] Standard cardinality estimation techniques help decide when to do this; our prototype does not implement cardinality estimation yet.

Multithreaded Execution. On a single server, our prototype runs a configurable number of executors over 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 data processing 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.

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 [35], and allows more optimization than `setjmp/longjmp` (SJLJ) intrinsics [36].

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. It also supports simple list comprehensions, control flow, random number generation, and regular expressions. For unsupported language features, Tuplex falls back on running the UDF in the Python interpreter. We believe that support for all 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 [49], Numba [31] or Bohrium [30] should be feasible in future work.

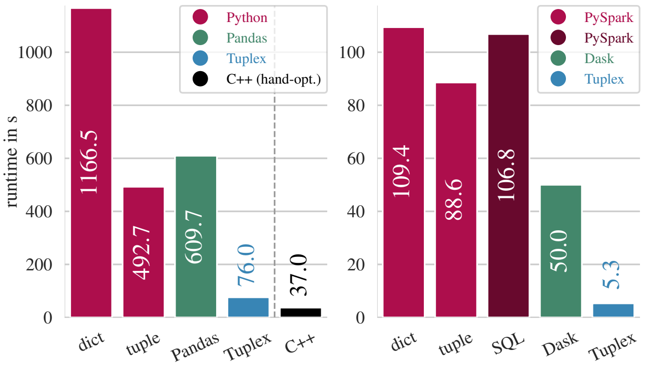
6 Evaluation

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? (§6.1)
2. 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 [49] and Hyper [26]? (§6.2)
3. What factors affect Tuplex's performance, and what is the impact of optimizations enabled by Tuplex's dual-mode processing model? (§6.3)
4. How does Tuplex perform when operating distributedly across many servers? (§6.4)

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 2: Dataset overview (smaller join tables excluded).



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

Figure 3: Tuplex outperforms single-threaded and parallel alternatives by 6.5×–20.6× when running the Zillow pipeline over 10G of input data, and comes close to hand-tuned C++.

Setup. In most experiments, Tuplex and other systems run on a single eight-socket server with 64 physical cores and 128 hyperthreads (8x Xeon E7-8830, 2.13GHz), 512 GB RAM, and four 1 TB HDDs configured in a RAID-0 setup. The input data is CSV-formatted UTF-8 text. We compare our Tuplex prototype against Dask v2.12.0 and Spark (Pyspark, v2.4.5) on Ubuntu 18.04. All systems use 16-way parallelism, and we pin them to the first two NUMA nodes on the machine to reduce variance. Numbers reported are the average of 10 runs with warmed up OS caches.

Our focus is on Tuplex's performance on a single multicore server, a common medium-scale analytics setup [12]. But the systems we compare against support scale-out across servers, and for this reason we also compare Tuplex's prototype AWS Lambda backend to Spark (§6.4).

6.1 End-to-End Performance

We measure Tuplex's end-to-end performance using three data science pipelines, and with the datasets shown in Table 2. The full pipelines are included in our supplementary material.

Zillow. Zillow is a real estate directory website whose listings are uploaded by individual brokers. We scraped 34,603 Boston area listings [56], scaled the data to 10 GB, and cleaned it for performance experiments to avoid failures in Spark and Dask. The query extracts information like the number of bedrooms, bathrooms, and the price from textual data.

It involves twelve 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 [17] and extended it by joining with additional airport and airline data from other sources (743 KB [51] and 82 KB [62]). Besides conversions, this query involves 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 query on ten years (2009-2019) of CSV data [63].

Weblogs. Based on a Spark use case [7], this query 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 data preparation query by an inner join with a list of bad IPs [43] and anonymize any personally-identifiable URLs by replacing usernames (*e.g.*, "`~alice`") with random strings.

311 and TPC-H Q6. We use the Pandas cookbook [13] data cleaning query for 311 service requests, which yields a set of unique zipcodes, to compare to Weld [49]. Finally, we also run a microbenchmark with TPC-H Q6 to measure Tuplex's performance compared to Hyper [26], a state-of-the-art SQL query compiler.

6.1.1 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), which allow lookups by column name. The dictionary representation eases implementation, but typically comes with a performance overhead. Tuplex allows either representation and compiles both representations into identical native code.

Single-threaded execution. We compare standard CPython (v3.6), Pandas (v1.0.1), and a hand-optimized C++ baseline 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. A good Tuplex implementation should come close to the hand-optimized C++ baseline.

Figure 3 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 23.7% slower than CPython. While Pandas benefits from a faster CSV parser, an efficient data representation for each column, and specialized native-code operators for numeric computation, its performance suffers when UDFs—for which Pandas has no efficient native operators—require processing in Python. Finally, Tuplex completes processing in 75.13 seconds, a speedup of 6.5×–15.5× over the CPython and Pandas implementations, and half as fast as the hand-optimized C++ implementation. However, this overstates Tuplex's overhead: in Tuplex the compute part of the pipeline (*i.e.*, excluding I/O) takes 20.6s, 22% slower than in the C++ implementation (16s).

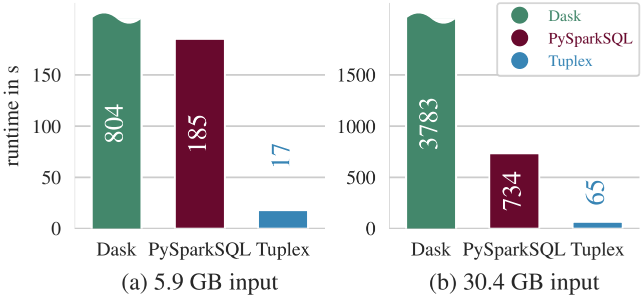


Figure 4: Tuplex achieves speedups of 10.9×–58.2× over PySparkSQL and Dask on the flights pipeline.

Data-parallel execution. Next, we benchmark Tuplex against widely-used frameworks for parallel processing of large inputs: PySpark (2.4.5) and Dask (2.16). We configure each system for 16-way parallelism: PySpark uses 16 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 [70] and with the more efficient SparkSQL operators [3]. Neither PySpark nor Dask compile UDFs to native code or optimize across UDFs, so a good result would show Tuplex outperforming them.

Figure 3 confirms that this is the case: Tuplex outperforms the fastest PySpark setup by 16.7× and Dask by 9.4×. Compared to the single-threaded execution, Tuplex achieves a speedup of 14.35× when using 16 threads.

These results confirm that Tuplex's native code generation for UDFs and its end-to-end optimization can offer performance gains for UDF-heavy pipelines. In §6.2.1, we compare Tuplex to other Python compilers, and §6.3 drills down into the factors contributing to Tuplex's performance.

6.1.2 Flights: Joins and Null Values.

We repeat the comparison between Tuplex, Spark, and Dask for the flights query. The flight query 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

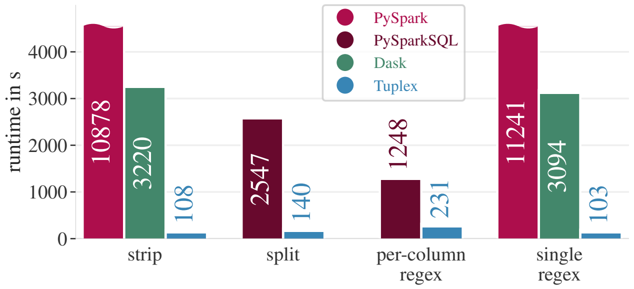


Figure 5: Tuplex outperforms Spark and Dask by 5.4×–109× on the weblogs pipeline; all Tuplex variants have comparable performance. PySparkSQL only supports per-column regexes.

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. A good result for Tuplex would show that it still outperforms the other systems, even though it must resolve some rows via the slower general-case path.

Figure 4 shows the results for two years' worth of data (5.9 GB) and ten years (30.4 GB). PySparkSQL outperforms Dask by 4–5× because of its compiled query plan and more efficient join operator. Tuplex, however, still achieves a 4.4× speedup over PySparkSQL (18.9× 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 §6.3.2). Tuplex's join operator is still unoptimized, so these results may understate its performance rather.

6.1.3 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 query splits an input log line into columns, and then rewrites one of those columns with a random string if it matches a username pattern:

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

While this step requires a UDF in all systems, 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 to the `split`-based UDF using a project operator that applies SparkSQL's native string functions (which execute in the JVM). In addition to these, PySparkSQL also has a native operator for regular expressions (`regexp_extract`). While it only supports per-column regular expressions (second setting), the operator applies the regular expression in the JVM, rather than in the Python interpreter. Finally, all systems currently require UDFs when using a *single* regular expression.[3] 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 an unfixed bug in the inner join, which fails when chunks produce empty results [64].

Figure 5 reports the results organized by setting. The PySpark pipelines with two UDFs are slowest at about 3 hours, while Dask UDFs are roughly 3× faster (51–53 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 21 (regex) from 42 minutes (split). This happens because SparkSQL can generate JVM bytecode for most of the pipeline (except the randomization UDF) via its whole-stage code generation [68]. Tuplex, on the other hand, completes the pipeline in two minutes 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 5.4×; likewise, Tuplex's `split`-based pipeline is 18.2× 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 3.5× 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 12.5× faster than `java.util.regex`, which Spark uses.

Tuplex's fastest pipelines (single regex, strip) outperform the best PySpark and Dask setups by 12× and 30×. Tuplex supports logical optimizations unavailable to Dask and Spark that improve performance further, which we discuss in §6.3.1.

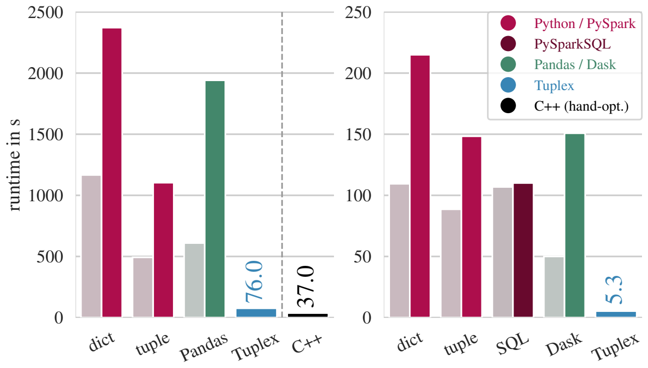
6.2 Comparison To Other Systems

We now compare Tuplex to prior systems that compile general Python programs, data science workloads, or SQL queries.

6.2.1 Python Compilers.

We first compare Tuplex to general Python compilers, which compile arbitrary Python programs.

[3] A PySparkSQL PR introducing support for a single regex returning multiple columns exists [15], but it did not work for us.



(a) single-threaded

(b) 16x parallelism

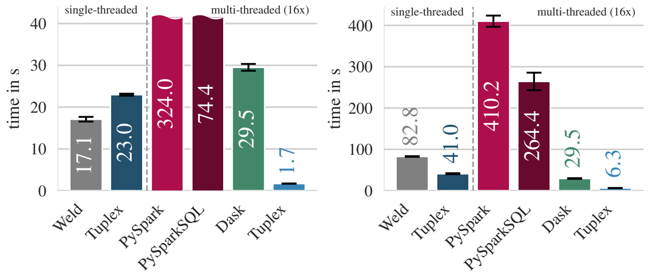
Figure 6: The PyPy3 general-purpose JIT fails to accelerate the Zillow pipeline, and degrades performance by up to 3×. Dark bars use PyPy, light bars use the CPython interpreter (Fig. 3).

	System	Runtime	Compile time
CPython (interpreter)		492.7 s	–
Python compilers	Cython	394.1 s	8.5 s
	Nuitka	386.5 s	5.3 s
	Tuplex	74.6 s	**0.6** s
Hand-optimized C++		**36.6** s	7.5 s

Figure 7: Tuplex runs the Zillow query 5× faster than Cython and Nuitka, and compiles 5×–10× faster than alternatives.

PyPy. PyPy [54] 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 (v7.3.0 in JIT mode) 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 6 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 [14] suggests that PyPy has a variable impact on UDF performance: of twelve UDFs, four are faster (25%–6×) with PyPy, four are up to 20% slower, and four are 50%–2.5× slower. The one UDF that benefits substantially (6×) merely forms a tuple; for 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 [60].



(a) query-time only

(b) end-to-end

Figure 8: 311 service requests cleaning.

Figure 9: For the 311 data cleaning pipeline, single-threaded Tuplex comes within 34% of Weld and outperforms all parallel systems. Tuplex outperforms Weld by 2× end-to-end because Tuplex inlines the aggregation in its generated parser.

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 (v0.6.9) and Cython (v0.29.16) to transpile the Python module to C for the Zillow pipeline. 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.

Figure 7 shows runtimes and compile times. Nuitka and Cython's compiled code runs 20% faster than interpreted Python, but is still over 5× 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 5–8× 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 7.4). Tuplex generates LLVM IR, which is faster to compile than higher-level C/C++, and also compiles 10× faster than gcc compiles the C++ baseline.

6.2.2 Data-parallel IR: Weld [49].

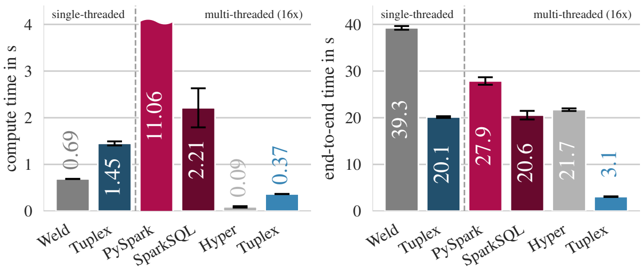
Weld is a data-parallel IR that admits optimizations like vectorization or loop fusion across libraries [49]. Weld serves as a backend to ported existing libraries such as Pandas [37], while Tuplex is a complete data analytics system, but both execute compiled native code. We compare Tuplex's performance to Weld's on the **311** data cleaning workload [13] and **TPC-H Q6**. Q6 performs a simple aggregation over numeric columns and is 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 [48], we compare single-threaded performance. In addition, we preprocessed the inputs to contain only the required columns and converted string-typed date columns in Q6 to integers, since Weld lacks automatic projection pushdown and has limited string processing capabilities. Because Weld does not have a native CSV parser, we preload the Q6 data into its columnar in-memory format with a single-threaded C++ CSV parser. For the 311 workload, we use Weld's benchmark code, which relies on Pandas to load the data. We measure both pure compute time and end-to-end runtime: the former measures how good Tuplex's generated code is, while the latter measures a realistic data analytics experience. A good result for Tuplex would show competitive compute time (despite Tuplex's row-oriented data format and lack of SIMD vectorization), and an improved end-to-end runtime.

Figure 9 shows that Tuplex's compute time (including compilation and sampling) for the 311 data cleaning workload is within 35% 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 10a), and Tuplex again outperforms Weld by 1.95× end-to-end (Figure 10b). 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.

6.2.3 SQL query compiler: Hyper [26].

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 [26, 45] to establish a baseline for Tuplex's performance on classic SQL queries. We use Tableau's latest HyperAPI [34] (0.0.11355) to run TPC-H Q6 with 16 threads. Hyper relies on indexes for performance [42]: 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 4.11× slower than Hyper's index-based execution (Figure 10a). 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 10b shows that Tuplex outperforms Hyper by 5.92× on end-to-end runtime, since Tuplex avoids upfront index creation and interleaves the aggregation with data loading through its generated parser.



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

Figure 10: For TPC-H Q6 Tuplex's generated code (without vectorization or indexes) comes within 2× of Weld's vectorized code and within 4× of Hyper's index-based execution. End-to-end, Tuplex outperforms Weld by 2× (due to its generated parser) and Hyper by 7× (as it avoids index creation).

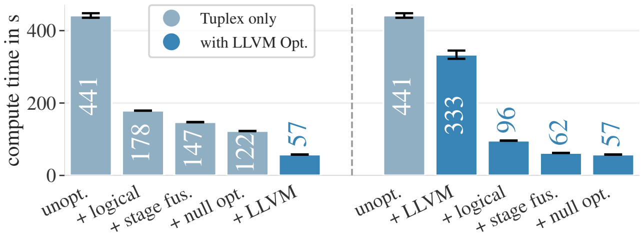


Figure 11: Factor analysis for the flights pipeline: Tuplex optimization and LLVM optimizers together realize speedups.

6.2.4 Discussion.

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 `-fastmath`). Further, Tuplex produces robust generated code that still contains instructions to check for exceptions, while Weld and Hyper only work on perfectly clean data.

6.3 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×.

We measure the impact of specific design choices and optimizations with the flights pipeline, using 4-way concurrency on a single NUMA node to avoid confounding factors (our prototype is not NUMA-aware yet). Figure 11 summarizes the impact of each factor on **flights** (30.4 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.

6.3.1 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 2.5× 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 the `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 55 seconds (2× faster than the unreordered result we reported in Figure 5). If we manually reorder the operators in PySparkSQL, it also runs 2× faster (611 seconds), but remains 14.5× slower than Tuplex.

6.3.2 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 seeks to make stages—which are its 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. §4.5). For each, we disable and enable LLVM optimizers to measure any extra cross-UDF optimization opportunities enabled. Without LLVM optimizers, Tuplex takes 178 seconds without stage fusion and 147 seconds with stage fusion (17.4% improvement); with LLVM optimizers, runtimes drop to 96 and 62 seconds (35.4% improvement). Stage fusion therefore enables optimization potential that improves runtime by an extra 18%.

6.3.3 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 (§4.7). In **flights**, this optimization reduces the pipeline's compute time by 8–17%, 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.) In **Zillow**, the end-to-end impact is 5% runtime reduction.

Setup	Spark (64 executors)	Tuplex (64 Lambdas)
100 GB	209.03 sec ($\sigma = 10.53$)	31.5 sec ($\sigma = 8.25$)
1 TB	1791.51 sec ($\sigma = 4.38$)	351.1 sec ($\sigma = 22.1$)

Figure 12: In a distributed scale-out experiment, Tuplex's Lambda backend outperforms a Spark cluster by 5.1×–6.6×.

6.4 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, whereas Tuplex's Lambda backend writes to S3, Spark merely collects results on its driver node, as writing to S3 requires extra infrastructure [23, 65]. We run the Zillow pipeline over scaled 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). A good result for Tuplex would show competitive performance, with its compiled UDFs amortizing the overheads incurred by Lambdas.

Figure 12 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 has the potential to be competitive in scale-out settings as well as on a single server.

7 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 [25, 52]. 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 in Tuplex, PySpark, and Dask, we found Tuplex preferable for several reasons:

- 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.
- Representing rows as tuples rather than dictionaries yields better PySpark performance, but the required numerical indexing took painstaking work to get right. Tuplex, by contrast, has identical performance for tuples and dictionaries, avoiding the speed-usability tradeoff.
- 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.
- 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). Our weblog dataset has two anomalous rows, which caused SparkSQL to silently return incorrect results, while Tuplex correctly reported the failed rows.
- We compared to Weld using the Pandas cookbook's subsampled 311 dataset [13] (99k rows) scaled 2,000× in §6.2.2, but Tuplex also works out-of-the-box on the full NYC 311 dataset [47] (22.4M rows), while Weld, PySpark, PySparkSQL, and Dask all fail and require changes to the UDF code for this more complex but realistic dataset.

We found that 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 quickly and accurately track down bugs in our pipelines.

Tuplex's dual mode processing works as long as the sample is representative. Like with any sampling approach, an unrepresentative sample can lead Tuplex to deduce an incorrect common case. In addition, the sample might itself produce only exceptions. If that is the case, Tuplex warns the user either to revise the pipeline or increase the sample size.

8 Conclusion

We introduced Tuplex, an new data analytics framework that compiles Python UDFs to optimized, native code. Tuplex's key idea is dual-mode processing, which makes optimizing UDF compilation tractable because it specializes the normal-case code to the common-case input data, and defers the complexity of handling other cases to a less-optimized exception-case code path. Our experiments show that Tuplex achieves speedups of 5–109× over Python, Pandas, Spark, and Dask, and comes close to hand-optimized C++ code. Tuplex's failure-free execution and exception resolution capabilities may also ease pipeline development.

Tuplex will be available as open-source software.

References

[1] 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.

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

[3] 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, pages 1383–1394.

[4] 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 (2011), pages 31 –39.

[5] 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*. PLDI '89. Portland, Oregon, USA: Association for Computing Machinery, 1989, 146–160.

[6] 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), pages 1466–1477.

[7] 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).

[8] Databricks, Inc. *Handling bad records and files*. URL: https://docs.databricks.com/spark/latest/spark-sql/handling-bad-records.html (visited on 05/26/2020).

[9] James M Decker, Dan Moldovan, Andrew A Johnson, Guannan Wei, Vritant Bhardwaj, Gregory Essertel, and Fei Wang. *Snek: Overloading Python Semantics via Virtualization*.

[10] 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)*. 2015, pages 41–48.

[11] 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*. ACM. 2013, pages 1–10.

[12] 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, pages 799–815.

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

[14] Python Software Foundation. *The Python Profilers: cProfile*. URL: https://docs.python.org/3/library/profile.html#module-cProfile (visited on 05/15/2020).

[15] Jiaan Geng. *[SPARK-24884][SQL] Support regexp function regexp_extract_all*. Feb. 2020. URL: https://github.com/apache/spark/pull/27507 (visited on 04/21/2020).

[16] *GraalVM*. https://www.graalvm.org/. (Accessed on 04/17/2019).

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

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

[19] 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).

[20] 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), pages 1256–1267.

[21] 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: *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 2019, pages 453–468.

[22] 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), pages 26–33.

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

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

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

[26] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots". In: *2011 IEEE 27th International Conference on Data Engineering*. IEEE. 2011, pages 195–206.

[27] 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), pages 2209–2222.

[28] 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.

[29] Sanjay Krishnan, Michael J. Franklin, Kenneth Goldberg, and Eugene Wu. "BoostClean: Automated Error Detection and Repair for Machine Learning". In: (Nov. 2017).

[30] 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*. PACT ’16. Haifa, Israel: Association for Computing Machinery, 2016, 71–85.

[31] 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: ACM, 2015, 7:1–7:6.

[32] 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].

[33] Chris Leary and Todd Wang. “XLA: TensorFlow, compiled”. In: *TensorFlow Dev Summit* (2017).

[34] Tableau Software LLC. *Tableau Hyper API*.

[35] 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).

[36] 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).

[37] 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).

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

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

[40] 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/.

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

[42] 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), pages 1702–1713.

[43] Myip.ms. *Blacklist IP Addresses Live Database (Realtime)*. https://myip.ms/browse/blacklist.

[44] 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*. 2019, pages 538–553.

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

[46] 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), pages 56–69.

[47] NYC OpenData. *311 Service Requests from 2010 to Present*.

[48] Shoumik Palkar.

[49] 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: volume 11. 9. VLDB Endowment, May 2018, pages 1002–1015.

[50] 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, pages 291–305.

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

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

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

[54] Armin Rigo, Maciej Fijalkowski, Carl Friedrich Bolz, Antonio Cuni, Benjamin Peterson, Alex Gaynor, Hakan Ardoe, Holger Krekel, and Samuele Pedroni. *PyPy*. URL: http://pypy.org/.

[55] 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". In: *arXiv preprint arXiv:1805.00907* (2018).

[56] ScrapeHero. *How to Scrape Real Estate Listings from Zillow.com using Python and LXML*. https://www.scrapehero.com/how-to-scrape-real-estate-listings-on-zillow-com-using-python-and-lxml/. Oct. 2017.

[57] 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*. SIGMOD '16. San Francisco, California, USA, 2016, 1907–1922.

[58] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. "Vectorization vs. Compilation in Query Execution". In: *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. DaMoN '11. Athens, Greece: Association for Computing Machinery, 2011, 33–40.

[59] 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*. SIGMOD '18. Houston, TX, USA, 2018, 307–322.

[60] The PyPy Team. *PyPy: Performance*. URL: https://www.pypy.org/performance.html (visited on 11/15/2019).

[61] *The Truffle Language Implementation Framework*. http://www.ssw.uni-linz.ac.at/Research/Projects/JVM/Truffle.html. (Accessed on 04/17/2019).

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

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

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

[65] 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*. SoCC '17. Santa Clara, California: Association for Computing Machinery, 2017, page 653.

[66] 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, pages 469–480.

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

[68] Reynold Xin and Josh Rosen. *Project Tungsten: Bringing Apache Spark Closer to Bare Metal*. 2015. URL: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[69] 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.

[70] 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*. USENIX Association. 2012, pages 2–2.

[71] 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, pages 1349–1350.

A Supplementary Material

A.1 Zillow Pipeline

```
def extractBd(x):
    val = x['facts and features']
    max_idx = val.find(' bd')
    if max_idx < 0:
        max_idx = len(val)
    s = val[:max_idx]

    # find comma before
    split_idx = s.rfind(',')
    if split_idx < 0:
        split_idx = 0
    else:
        split_idx += 2
    r = s[split_idx:]
    return int(r)

def extractBa(x):
    val = x['facts and features']
    max_idx = val.find(' ba')
    if max_idx < 0:
        max_idx = len(val)
    s = val[:max_idx]

    # find comma before
    split_idx = s.rfind(',')
    if split_idx < 0:
        split_idx = 0
    else:
        split_idx += 2
    r = s[split_idx:]
    return int(r)

def extractSqft(x):
    val = x['facts and features']
    max_idx = val.find(' sqft')
    if max_idx < 0:
        max_idx = len(val)
    s = val[:max_idx]

    split_idx = s.rfind('ba ,')
    if split_idx < 0:
        split_idx = 0
    else:
        split_idx += 5
    r = s[split_idx:]
    r = r.replace(',', '')
    return int(r)

def extractOffer(x):
    offer = x['title'].lower()
    if 'sale' in offer:
        return 'sale'
    if 'rent' in offer:
        return 'rent'
    if 'sold' in offer:
        return 'sold'
    if 'foreclose' in offer.lower():
        return 'foreclosed'
    return offer

def extractType(x):
    t = x['title'].lower()
    type = 'unknown'
    if 'condo' in t or 'apartment' in t:
        type = 'condo'
    if 'house' in t:
        type = 'house'
    return type

def extractPrice(x):
    price = x['price']
    p = 0
    if x['offer'] == 'sold':
        # price is to be calculated using price/sqft * sqft
        val = x['facts and features']
        s = val[val.find('Price/sqft:') + len('Price/sqft:') + 1:]
        r = s[s.find('$')+1:s.find(', ') - 1]
        price_per_sqft = int(r)
        p = price_per_sqft * x['sqft']
    elif x['offer'] == 'rent':
        max_idx = price.rfind('/')
        p = int(price[1:max_idx].replace(',', ''))
    else:
        # take price from price column
        p = int(price[1:].replace(',', ''))

    return p

def filterPrice(x):
    return 100000 < x['price'] <= 2e7

def filterType(x):
    return x['type'] == 'house'

def filterBd(x):
    return x['bedrooms'] < 10


ctx = tuplex.Context()

ctx.csv(','.join(paths)) \
    .withColumn("bedrooms", extractBd) \
    .filter(lambda x: x['bedrooms'] < 10) \
    .withColumn("type", extractType) \
    .filter(lambda x: x['type'] == 'house') \
    .withColumn("zipcode", lambda x: '%05d' % int(x['postal_code'])) \
    .mapColumn("city", lambda x: x[0].upper() + x[1:].lower()) \
    .withColumn("bathrooms", extractBa) \
    .withColumn("sqft", extractSqft) \
    .withColumn("offer", extractOffer) \
    .withColumn("price", extractPrice) \
    .filter(lambda x: 100000 < x['price'] < 2e7) \
    .selectColumns(["url", "zipcode", "address", "city", "state",
                    "bedrooms", "bathrooms", "sqft", "offer", "type", "price"]) \
    .tocsv(output_path)
```

A.2 Flights Pipeline

We disabled the reordering optimization in the evaluation section, though it is also possible for this pipeline.

```
carrier_hist_path = 'data/L_CARRIER_HISTORY.csv'
airport_data_path = 'data/GlobalAirportDatabase.txt'

ctx = tuplex.Context(conf)

df = ctx.csv(','.join(perf_paths))

renamed_cols = list(map(lambda c: ''.join(map(lambda w: w.capitalize(), c.split('_'))), df.columns))
for i, c in enumerate(df.columns):
    df = df.renameColumn(c, renamed_cols[i])

df = df.withColumn('OriginCity', lambda x: x['OriginCityName'][:x['OriginCityName'].rfind(',')].strip())
df = df.withColumn('OriginState', lambda x: x['OriginCityName'][x['OriginCityName'].rfind(',')+1:].strip())
df = df.withColumn('DestCity', lambda x: x['DestCityName'][:x['DestCityName'].rfind(',')].strip())
df = df.withColumn('DestState', lambda x: x['DestCityName'][x['DestCityName'].rfind(',')+1:].strip())
df = df.mapColumn('CrsArrTime', lambda x: '{:02}:{:02}'.format(int(x / 100), x % 100) if x else None)
df = df.mapColumn('CrsDepTime', lambda x: '{:02}:{:02}'.format(int(x / 100), x % 100) if x else None)

def cleanCode(t):
    if t["CancellationCode"] == 'A':
        return 'carrier'
    elif t["CancellationCode"] == 'B':
        return 'weather'
    elif t["CancellationCode"] == 'C':
        return 'national air system'
    elif t["CancellationCode"] == 'D':
        return 'security'
    else:
        return None

def divertedUDF(row):
    diverted = row['Diverted']
    ccode = row['CancellationCode']
    if diverted:
        return 'diverted'
    else:
        if ccode:
            return ccode
        else:
            return 'None'

def fillInTimesUDF(row):
    ACTUAL_ELAPSED_TIME = row['ActualElapsedTime']
    if row['DivReachedDest']:
        if float(row['DivReachedDest']) > 0:
            return float(row['DivActualElapsedTime'])
        else:
            return ACTUAL_ELAPSED_TIME
    else:
        return ACTUAL_ELAPSED_TIME

df = df.withColumn('CancellationCode', cleanCode) # works...
df = df.mapColumn('Diverted', lambda x: True if x > 0 else False)
df = df.mapColumn('Cancelled', lambda x: True if x > 0 else False)

df = df.withColumn('CancellationReason', divertedUDF)
df = df.withColumn('ActualElapsedTime', fillInTimesUDF)

df_carrier = ctx.csv(carrier_hist_path)

def extractDefunctYear(t):
    x = t['Description']
    desc = x[x.rfind('-') + 1:x.rfind(')')].strip()
    return int(desc) if len(desc) > 0 else None

df_carrier = df_carrier.withColumn('AirlineName', lambda x: x['Description'][:x['Description'].rfind('(')].strip())
df_carrier = df_carrier.withColumn('AirlineYearFounded', lambda x: int(x['Description'][x['Description'].rfind('(') + 1:x['Description'].rfind('-')]))
df_carrier = df_carrier.withColumn('AirlineYearDefunct', extractDefunctYear)

airport_cols = ['ICAOCode', 'IATACode', 'AirportName', 'AirportCity', 'Country',
                'LatitudeDegrees', 'LatitudeMinutes', 'LatitudeSeconds', 'LatitudeDirection',
                'LongitudeDegrees', 'LongitudeMinutes', 'LongitudeSeconds',
                'LongitudeDirection', 'Altitude', 'LatitudeDecimal', 'LongitudeDecimal']

df_airports = ctx.csv(airport_data_path, columns=airport_cols, delimiter=':', header=False, null_values=['', 'N/a', 'N/A'])

df_airports = df_airports.mapColumn('AirportName', lambda x: string.capwords(x))
df_airports = df_airports.mapColumn('AirportCity', lambda x: string.capwords(x))

df_all = df.join(df_carrier, 'OpUniqueCarrier', 'Code')
df_all = df_all.leftJoin(df_airports, 'Origin', 'IATACode', prefixes=(None, 'Origin'))
df_all = df_all.leftJoin(df_airports, 'Dest', 'IATACode', prefixes=(None, 'Dest'))
df_all = df_all.mapColumn('Distance', lambda x: x / 0.00062137119224)

df_all = df_all.mapColumn('AirlineName', lambda s: s.replace('Inc.', '') \
                          .replace('LLC', '') \
                          .replace('Co.', '').strip())

df_all = df_all.renameColumn('OriginLongitudeDecimal', 'OriginLongitude') \
    .renameColumn('OriginLatitudeDecimal', 'OriginLatitude') \
    .renameColumn('DestLongitudeDecimal', 'DestLongitude') \
    .renameColumn('DestLatitudeDecimal', 'DestLatitude')

df_all = df_all.renameColumn('OpUniqueCarrier', 'CarrierCode') \
    .renameColumn('OpCarrierFlNum', 'FlightNumber') \
    .renameColumn('DayOfMonth', 'Day') \
    .renameColumn('AirlineName', 'CarrierName') \
    .renameColumn('Origin', 'OriginAirportIATACode') \
    .renameColumn('Dest', 'DestAirportIATACode')


# remove rows that make no sense, i.e. all flights where the airline is defunct which may happen after the join
def filterDefunctFlights(row):
    year = row['Year']
    airlineYearDefunct = row['AirlineYearDefunct']

    if airlineYearDefunct:
        return int(year) < int(airlineYearDefunct)
    else:
        return True

df_all = df_all.filter(filterDefunctFlights)

numeric_cols = ['ActualElapsedTime', 'AirTime', 'ArrDelay',
                'CarrierDelay', 'CrsElapsedTime',
                'DepDelay', 'LateAircraftDelay', 'NasDelay',
                'SecurityDelay', 'TaxiIn', 'TaxiOut', 'WeatherDelay']
```

```
for c in numeric_cols:
    df_all = df_all.mapColumn(c, lambda x: int(x) if x else 0)

df_all.selectColumns(['CarrierName', 'CarrierCode', 'FlightNumber',
                      'Day', 'Month', 'Year', 'DayOfWeek',
                      'OriginCity', 'OriginState', 'OriginAirportIATACode', 'OriginLongitude', 'OriginLatitude',
                      'OriginAltitude',
                      'DestCity', 'DestState', 'DestAirportIATACode', 'DestLongitude', 'DestLatitude', 'DestAltitude',
                      'Distance',
                      'CancellationReason', 'Cancelled', 'Diverted', 'CrsArrTime', 'CrsDepTime',
                      'ActualElapsedTime', 'AirTime', 'ArrDelay',
                      'CarrierDelay', 'CrsElapsedTime',
                      'DepDelay', 'LateAircraftDelay', 'NasDelay',
                      'SecurityDelay', 'TaxiIn', 'TaxiOut', 'WeatherDelay',
                      'AirlineYearFounded', 'AirlineYearDefunct']).tocsv(output_path)
```

A.3 Weblogs Pipeline

A.3.1 strip

```
def ParseWithStrip(x):
    y = x

    i = y.find(" ")
    ip = y[:i]
    y = y[i + 1 :]

    i = y.find(" ")
    client_id = y[:i]
    y = y[i + 1 :]

    i = y.find(" ")
    user_id = y[:i]
    y = y[i + 1 :]

    i = y.find("]")
    date = y[:i][1:]
    y = y[i + 2 :]

    y = y[y.find('"') + 1 :]

    method = ""
    endpoint = ""
    protocol = ""
    failed = False
    if y.find(" ") < y.rfind('"'):
        i = y.find(" ")
        method = y[:i]
        y = y[i + 1 :]

        i = y.find(" ")
        endpoint = y[:i]
        y = y[i + 1 :]

        i = y.rfind('"')
        protocol = y[:i]
        protocol = protocol[protocol.rfind(" ") + 1 :]
        y = y[i + 2 :]
    else:
        failed = True
        i = y.rfind('"')
        y = y[i + 2 :]

    i = y.find(" ")
    response_code = y[:i]
    content_size = y[i + 1 :]

    if not failed:
        return {"ip": ip,
                "client_id": client_id,
                "user_id": user_id,
                "date": date,
                "method": method,
                "endpoint": endpoint,
                "protocol": protocol,
                "response_code": int(response_code),
                "content_size": 0 if content_size == '-' else int(content_size)}
    else:
        return {"ip": "",
                "client_id": "",
                "user_id": "",
                "date": "",
                "method": "",
                "endpoint": "",
                "protocol": "",
                "response_code": -1,
                "content_size": -1}

def randomize_udf(x):
    return re.sub('^/~[^/]+', '/~' + ''.join([random.choice('ABCDEFGHIJKLMNOPQRSTUVWXYZ') for t in range(10)]), x)

ctx = tuplex.Context()

df = ctx.text(','.join(paths)).map(ParseWithStrip).mapColumn("endpoint", randomize_udf)
bad_ip_df = ctx.csv(ip_blacklist_path)

df_malicious_requests = df.join(bad_ip_df, "ip", "BadIPs")
df_malicious_requests.selectColumns(["ip", "date", "method", "endpoint", "protocol", "response_code", "content_size"]).tocsv(output_path)
```

A.3.2 split

```
def randomize_udf(x):
    return re.sub('^/~[^/]+', '/~' + ''.join([random.choice('ABCDEFGHIJKLMNOPQRSTUVWXYZ') for t in range(10)]), x)

ctx = tuplex.Context()
df = (
    ctx.text(','.join(paths))
    .map(lambda x: {'logline': x})
    .withColumn("cols", lambda x: x['logline'].split(' '))
    .withColumn("ip", lambda x: x['cols'][0].strip())
    .withColumn("client_id", lambda x: x['cols'][1].strip())
    .withColumn("user_id", lambda x: x['cols'][2].strip())
    .withColumn("date", lambda x: x['cols'][3] + " " + x['cols'][4])
    .mapColumn("date", lambda x: x.strip())
    .mapColumn("date", lambda x: x[1:-1])
    .withColumn("method", lambda x: x['cols'][5].strip())
    .mapColumn("method", lambda x: x[1:])
    .withColumn("endpoint", lambda x: x['cols'][6].strip())
    .withColumn("protocol", lambda x: x['cols'][7].strip())
    .mapColumn("protocol", lambda x: x[:-1])
    .withColumn("response_code", lambda x: int(x['cols'][8].strip()))
    .withColumn("content_size", lambda x: x['cols'][9].strip())
    .mapColumn("content_size", lambda x: 0 if x == '-' else int(x))
    .filter(lambda x: len(x['endpoint']) > 0)
    .mapColumn("endpoint", randomize_udf)
)

bad_ip_df = ctx.csv(ip_blacklist_path)

df_malicious_requests = df.join(bad_ip_df, "ip", "BadIPs")
df_malicious_requests.selectColumns(["ip", "date", "method", "endpoint", "protocol", "response_code", "content_size"]).tocsv(output_path)
```

A.3.3 single-regex

```
def ParseWithRegex(logline):
    match = re.search('^(\S+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(\S+) (\S+)\s*(\S*)\s*" (\d{3}) (\S+)', logline)
    if(match):
        return {"ip": match[1],
                "client_id": match[2],
                "user_id": match[3],
                "date": match[4],
                "method": match[5],
                "endpoint": match[6],
                "protocol": match[7],
                "response_code": int(match[8]),
                "content_size": 0 if match[9] == '-' else int(match[9])}
    else:
        return {"ip": '',
                "client_id": '',
                "user_id": '',
                "date": '',
                "method": '',
                "endpoint": '',
                "protocol": '',
                "response_code": -1,
                "content_size": -1}

def randomize_udf(x):
    return re.sub('^/~[^/]+', '/~' + ''.join([random.choice('ABCDEFGHIJKLMNOPQRSTUVWXYZ') for t in range(10)]), x)
```

```
ctx = tuplex.Context()

df = ctx.text(','.join(paths)).map(ParseWithRegex).mapColumn("endpoint", randomize_udf)
bad_ip_df = ctx.csv(ip_blacklist_path)

df_malicious_requests = df.join(bad_ip_df, "ip", "BadIPs")
df_malicious_requests.selectColumns(["ip", "date", "method", "endpoint", "protocol", "response_code", "content_size"]).tocsv(output_path)
```

B UDF Example

Here, we show the resulting LLVM IR from the introductory example `lambda m: m * 1.609 if m else 0.0` in Section 2 after applying `-O3` passes. Note that Tuplex will inline functions and run additional optimization passes when these functions are used within a pipeline.

```
; ModuleID = 'sampleudf.ll'
source_filename = "tuplex"

%tuple = type { double }
%tuple.0 = type { i64 }
%tuple.1 = type { [1 x i1], i64 }
%tuple.2 = type {}

; Function generated when type is specialized
; to int64_t
define i64 @lami64(%tuple*  %outRow,
                 %tuple.0*  readonly byval %inRow) #1 {
body:
  %0 = getelementptr %tuple.0, %tuple.0* %inRow, i64 0, i32 0
  %1 = load i64, i64* %0, align 8
  %2 = icmp eq i64 %1, 0
  %3 = sitofp i64 %1 to double
  %4 = fmul double %3, 1.609000e+00
  %.0 = select i1 %2, double 0.000000e+00, double %4
  %5 = getelementptr %tuple, %tuple* %outRow, i64 0, i32 0
  store double %.0, double* %5, align 8
  ret i64 0
}

; Function generated when type is specialized
; to Nullable<int64_t>
define i64 @lamopti64(%tuple*  %outRow,
                   %tuple.1*  readonly byval %inRow) #1 {
body:
  %0 = getelementptr %tuple.1, %tuple.1* %inRow, i64 0, i32 0, i64 0
  %1 = load i1, i1* %0, align 1
  %2 = getelementptr %tuple.1, %tuple.1* %inRow, i64 0, i32 1
  %3 = load i64, i64* %2, align 8
  %4 = icmp ne i64 %3, 0
  %not. = xor i1 %1, true
  %spec.select = and i1 %4, %not.
  br i1 %spec.select, label %if, label %ifelse_exit

if:
  br i1 %1, label %except, label %next

ifelse_exit:
  %.0 = phi double [ %7, %next ], [ 0.000000e+00, %body ]
  %5 = getelementptr %tuple, %tuple* %outRow, i64 0, i32 0
  store double %.0, double* %5, align 8
  ret i64 0

next:
  %6 = sitofp i64 %3 to double
  %7 = fmul double %6, 1.609000e+00
  br label %ifelse_exit

except:
  ret i64 129
}

; Function generated when type is specialized
; to Null
define i64 @lamnone(%tuple*  %outRow,
                  %tuple.2*  readnone byval %inRow) #2 {
body:
  %0 = getelementptr  %tuple, %tuple* %outRow, i64 0, i32 0
  store double 0.000000e+00, double* %0, align 8
  ret i64 0
}
```