Tuplex: Robust, Efficient Analytics When Python Rules

Leonhard F. Spiegelberg
Brown University
leonhard_spiegelberg@brown.edu

Tim Kraska
Massachusetts Institute of Technology
kraska@mit.edu

ABSTRACT

Spark became the defacto industry standard as an execution engine for data preparation, cleaning, distributed machine learning, streaming and, warehousing over raw data. However, with the success of Python the landscape is shifting again; there is a strong demand for tools which better integrate with the Python landscape and do not have the impedance mismatch like Spark. In this paper, we demonstrate *Tuplex* (short for *tuples* and *exceptions*), a Python-native data preparation framework that allows users to develop and deploy pipelines faster and more robustly while providing bare-metal execution times through code compilation whenever possible.

PVLDB Reference Format:
Leonhard F. Spiegelberg and Tim Kraska. Tuplex: Robust, Efficient Analytics When Python Rules. *PVLDB*, 12(12): 1958-1961, 2019.
DOI: https://doi.org/10.14778/3352063.3352109

1. INTRODUCTION

Python won. It is today *THE* language for data science. It provides libraries for almost anything, is driven by a large community, and its ease of use, expressiveness and high-level features make it an extremely powerful language [7].

With Python becoming the leading data science language, millions of users are relying on a common data science stack consisting usually of Jupyter or Zeppelin notebooks with various packages/modules: PySpark to distributively prepare data by parsing, cleaning, and filtering; Numpy, Scipy, Pandas to explore (samples of) the data and to transform it; Scikit-learn allows to build (statistical) models over small data while PyTorch, TensorFlow, Keras, or CNTK are used to build models over large datasets.

However, PySpark stands out as it is the only framework, which was not designed from the beginning to integrate with Python as it is actually JVM-based. The latter is problematic in a world where Python dominates the Data Science landscape. In Spark Python support (similar to R support) was only added as an afterthought and does not achieve the same performance as other frameworks, which were designed with Python in mind. For example, Tensorflow, Numpy, Scikit-Learn, and Keras combine a Python frontend seamlessly with a C++ backend for performance and the possibility to take advantage of modern hardware (GPUs, TPUs, FPGAs). This combination works particularly well as it allows to transfer data between Python, other libraries and C++ without unnecessary copies, conversions and minimal communication overhead.

Moreover, from all the mentioned frameworks only Spark is a general purpose platform aiming to do it all from data integration, over analytics and streaming, to model building. However, according to the "one-size-does-not-fit-all" theory [10], often huge performance gains can be achieved by building more specialized systems [9].

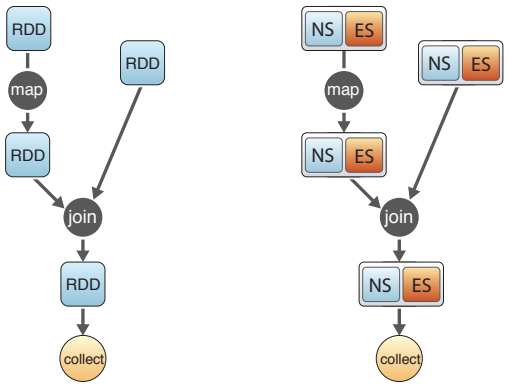
We therefore started to design *Tuplex* as an experiment to explore how a more efficient and Python-optimized framework for distributed data preparation could overcome some of Python's inherent problems. *Tuplex*, short for *tuples* and *exceptions*, provides bare-metal performance whenever possible through aggressive code generation similar to other Python-optimized frameworks (e.g., Tensorflow). However, what distinguishes *Tuplex* the most from Spark and other data preparation frameworks (e.g., Pandas) is the way *Tuplex* handles errors and its optimization for the common case. For example, envision the task of parsing a column in a CSV file into an array of integers. While this can be very efficiently expressed as a single map function in Python, exceptions are usually the problem; e.g., the one row in the CSV file which has the value "N/A" in it rather than a valid integer and causes the program to crash. Such cases are omnipresent in any real data preparation pipeline and *Tuplex* offers a simple and elegant solution to the problem, which allows for faster prototyping. Furthermore, as we will show, our new processing model enables a whole new set of additional optimizations.

In the following, we first describe the exception-aware processing model of *Tuplex*, then discuss the current implementation and outline various optimizations *Tuplex* does, and finally describe the demo scenario.

2. TUPLEX PROCESSING MODEL

It is very common that a data preparation pipeline running over the actual data for the first time will crash. This might be because of misalignment (e.g., a CSV row is not complete), null values, wrong types (e.g., a string instead of

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352109



(a) Spark **(b)** Tuplex

Figure 1: Spark uses a chain of RDDs triggered by an action (here: collect). One or more RDDs produce another RDD. Tuplex uses a chain of normal sets(NS), with each normal set potentially producing an exception set(ES) that is merged back whenever resolution is possible.

an integer), extreme values (e.g., numeric overruns), problems in the previous transformation step, among many other reasons. *Tuplex* wants to avoid such crashes and the processing model was particularly designed based on the observation that (1) developers want to have a first working end-to-end pipeline as fast as possible, (2) errors are better resolved in an iterative manner later on, (3) it is often better to collect all errors rather than crash on the first occurrence, and (4) that in some cases those errors do not matter much. We therefore developed a simple but powerful variant of the Spark processing paradigm. Instead of having each operation take one or more RDDs (i.e., a distributed collection of data) as input and producing one RDD as output, *Tuplex*'s processing model makes exceptions explicit by having normal sets (NS) and exception sets (ES). Every operation in *Tuplex* takes one or more (NS, ES)-pairs, called Tuplex Pairs or *TP*s, as input and produces one TP pair as output:

$$\text{op} : (\text{NS}_1, \text{ES}_1), ..., (\text{NS}_n, \text{ES}_n) \rightarrow (\text{NS}, \text{ES})$$

A pipeline is composed of multiple operations forming a DAG. Similar to Spark, a DAG of operations is lazily executed by appending an action. An action is an operation that triggers computation and materializes data, e.g. to be saved to disk or to return a list of Python tuples. However, to ensure the API is compatible with the standard way users program today, *Tuplex* hides the ES input and makes it only accessible through its API. For example, the following code multiplies the value of the first column with 0.5. As one can

```
c.csv('/data/flights*.csv') \
 .map(lambda x: 0.5 * x[0] ) \
 .collect()
```

see the code shown below is the same as one would program today with PySpark or Python. *Tuplex* currently supports a similar API to Spark. However, there exists an important difference: whenever the *map* throws an exception for an input row, the according row is placed into the exception set. This exception set can then be resolved at the end of the program, by the next operation, or not be resolved at all. However, in any case, the user is warned by visual output if such an exception did happen.

For example, users can add special *resolve functions* to their pipelines like in the code shown below. The backend then automatically applies these functions to the rows causing exceptions with a matching exception type and ultimately merges these rows back to the error-free data (i.e. thus becoming part of a normal set again). Interesting

```
c.csv('/data/flights*.csv') \
 .map(lambda x: 0.5 * x[0] ) \
 .resolve(NotANumber, lambda x: 0.0) \
 .resolve(TypeError, lambda x: 0.0) \
 .collect()
```

from a systems perspective is, that all exceptions are effectively tracked through the entire pipeline. For example, *Tuplex* keeps the lineage information for every exception around. Furthermore, it allows to access the exception set at every stage in the pipeline or in aggregated form at the end. This has several advantages: (1) it allows to more efficiently debug an exception even if the execution is distributed, and (2) it allows to take all exceptions, move them into a normal set, and write specific code for them. The latter efficiently makes it possible to iteratively remove problem cases while having a first working pipeline as fast as possible.

Code for *Tuplex* is written using standard Python in the terminal, jupyter notebook or by running `.py`-files. *Tuplex* allows to track any Python exception thrown during any stage within the pipeline when executing compiled code or UDFs in fallback mode. The idea of saving exceptions is similar to load-reject files for classical DBMS loading, however, the difference is, that UDFs may throw errors at any stage during the pipeline which can not be tracked easily or even handled at all through traditional load-reject files. Furthermore, in the next section we describe how this mechanism can be leveraged to produce optimized code similar to speculative optimization in JIT engines [12] or classical profile guided optimization.

3. TUPLEX ARCHITECTURE

Our current *Tuplex* prototype, which implements the above process model, is implemented using Python 3.5+ for the frontend and C++11 for the backend, compiler and execution engine from scratch. The compiler frontend for Python is custom written and LLVM 5.0 is used as backend with its optimization capabilities. The execution engine currently runs multi-threaded on a single node and has a fallback mode using the cloudpickle module [8] for any Python function that we can not transform to more efficient LLVM code. That way users are not restricted in the libraries or code they write. In the following sections we describe some of *Tuplex*'s unique optimizations.

3.1 Smart Caching

When users develop big data pipelines, they often first sample the data to prototype and test both the pipeline with its UDFs before running it over the entire data. This process might then repeat over various samples until the pipeline is run over the entire data. With Spark, it is possible that a single tuple (e.g., in a aggregation) may crash the entire job after having spent thousands of CPU days, leaving users to debug logs of hundreds of nodes. However, our novel processing model not only avoids those crashes but also offers

several unique opportunities for caching. *Tuplex* caches results (normal and error set) between invocations and only executes what has changed. For example, if a user adds resolutions only the exceptions are reprocessed and not the entire pipeline. Note, that in any other framework adding an exception handler would actually cause the entire pipeline to be executed again over the entire dataset.

This simple optimization has shown to significantly reduce the overall execution and development time as it avoids expensive re-executions.

3.2 Optimizing for the Normal Case

While exceptions are common, relatively speaking they are rare. For example, it is reasonable to assume that most rows in a CSV are not broken but a few might be. Furthermore, users tend to test their pipelines on a small subset of the data first. This opens up new ways of optimization. Most importantly, we can use the sample to figure out the common case and optimize the code accordingly. Hence exceptions sets make it not only easier for the user to deal with exceptions, we can also use it to stage computation. That is, first we use a fast code-path for the most common case, which puts every item it can not handle in the exception set. We then (automatically) use a more general code to handle the exception cases, and finally create the exception set for those errors only the user can help with. We now provide a few examples of how to optimize for the normal case:

Typing for the Normal Case: Python uses a typing model known as duck typing [11] that does not check types at compile type but is strongly typed during runtime. This means that the same function can be used for strings, integers, floats or other suitable custom types. A general compiler must account for all these cases, making compilation difficult and often infeasible. However, at runtime, there are clearly defined type constraints for which operations can be performed. Thus, when processing data in a MapReduce fashion [3], input types can actually be inferred at query-compilation time to create a dataflow graph for UDF compilation.

However, sometimes it is not possible to compute a typed dataflow graph. Here, *Tuplex* uses a probabilistic approach for the common case and computes the *most likely* dataflow path in order to generate native code for it. Deviations from this path are then treated as exceptions for which automatic resolvers are generated. Hence, the *Tuplex* framework restricts the typing in the sense that each column of input data is assumed to adhere to one type normally, i.e. the *most likely* one, for the majority case of the data (cf. Figure 2). This might seem at first sight similar to speculative optimization in JIT engines, e.g. implemented in Java HotSpot or V8 for Javascript, however there some key differences: (1) first, sampling and compilation are done ahead of time. (2) second, instead of filling a feedback vector and checking dynamically whether optimized code can be executed, optimized code is always executed and whenever an error condition or deviation from the normal case is encountered an exception is thrown. (3) third, resolution is deferred until the normal case terminated and then attempted.

Compression for the Normal Case: The execution model of splitting the output of an operator into a normal and an exceptional set allows to introduce artificial exceptions for increased execution speed. For example, in Python, integers are by default represented as 64-bit values. Many

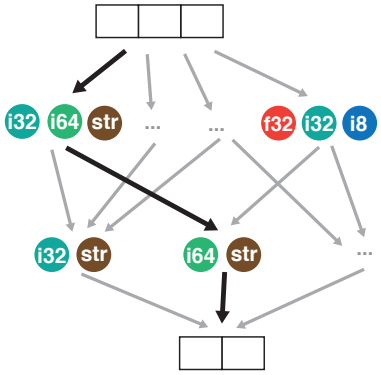


Figure 2: A dataflow graph computed from a sample of input data. The most likely datapath is shown together with the most likely typing along its edges. This path is used to generate native code for the normal path. When input deviating from the normal path is encountered, an exception is thrown and resolved after the normal case computed.

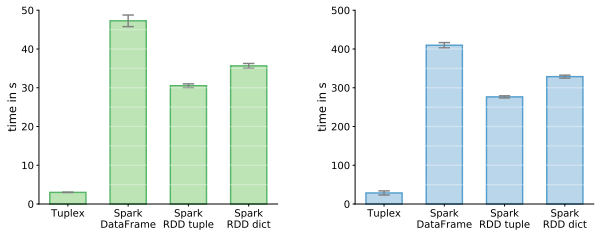
data fields, however, (e.g. a year, month, day column) require fewer bits, and computations over them may be expressed more efficiently as e.g. 16-bit integer operations. Because the program should semantically follow valid Python, in the case that an input value is found to require 64-bit computation, an artificial exception is thrown and a resolver is generated to compute its results using full 64-bit integer operations. The same approach can be used to compress constants and strings on-the-fly, or to perform delta-encoding on numeric values. Aside from leveraging compiler optimizations (e.g. peephole optimizations or auto-vectorization), this enables *Tuplex* to use a more efficient internal materialization format. Native code for a pipeline of Python UDFs is then generated based on ideas from Tupleware [2] and Hyper [5]. To speed up typical data preparation tasks over CSV files, *Tuplex* has a module to generate code for an efficient reader for CSV files according to the RFC-4180 standard. For this, ideas similar to Mison like aggressive column projection and speculation are used [4].

3.3 Initial Results

We tested *Tuplex* on a data integration scenario using scraped Zillow data for which a comparable Spark job fails. In order to show a 1:1 performance benchmark rows that would lead to an exception within a UDF were excluded from the original input file. The resulting file was then scaled to a size of 1GB and 10GB respectively. The query shown here consists of 12 UDFs written in Python to clean the data and outputs the result as a single CSV file. In a direct comparison depicted in Figure 3 *Tuplex* is around an order of magnitude faster than Spark 2.4.2. The query was implemented using Spark's DataFrame API [1] and in the older RDD API [13] using Python dictionaries or tuples to represent rows.

4. DEMONSTRATION OVERVIEW

In our demonstration, we want to show how users can easily build pipelines, track errors and resolve them using Python in interactive mode, file mode or via Jupyter notebooks [6] using the *Tuplex* framework and *Tuplex WebUI*. For this, we showcase a typical data scientist workflow where



(a) 1 GB **(b)** 10 GB

Figure 3: In this sample query Tuplex achieves a speedup of $\approx 9.7\times$ to $15.75\times$ over Spark 2.4.2. Startup time was excluded for Spark, whereas times for Tuplex are end-to-end. Each framework was configured to use 4 executors and a maximum of 16GB of main memory. The benchmark was performed on a AWS EC2 r5xlarge instance with a 150GB SSD drive, numbers are averaged over 6 runs.

scraped data from Zillow is analyzed. Users can experience the error tracking capabilities of *Tuplex* first-hand for this dataset and how the framework helps them to rapidly explore data using a Jupyter notebook without having to worry about breaking the pipeline. In a second example, we show the superior execution speed of *Tuplex* over Spark on a scaled out version of the (scraped) Zillow dataset we created. The sample query we demonstrate consists of simple UDFs extracting information, e.g. from a string like `'2 bds, 3 ba, 2,740+ sqft'`, type conversions, filtering and column selection. The *Tuplex WebUI* we will use to show the error cases is shown in Figure 4. It tracks any errors that occurred during execution (1) in real time. To help users to write code for error resolution, users are displayed a summary of errors (2) caused within an operator as well as a traceback for the first row (3) that caused an exception. In addition, for each exception type a small sample of the input data transformed by the operator is shown (4). Furthermore, users are able to investigate the physical plan the backend generates, and its performance (5) (e.g., how much time is spent on the normal sets and error sets).

Finally, we will show by means of example how *Tuplex* is able to optimize for the normal case and provide a direct comparison to Spark.

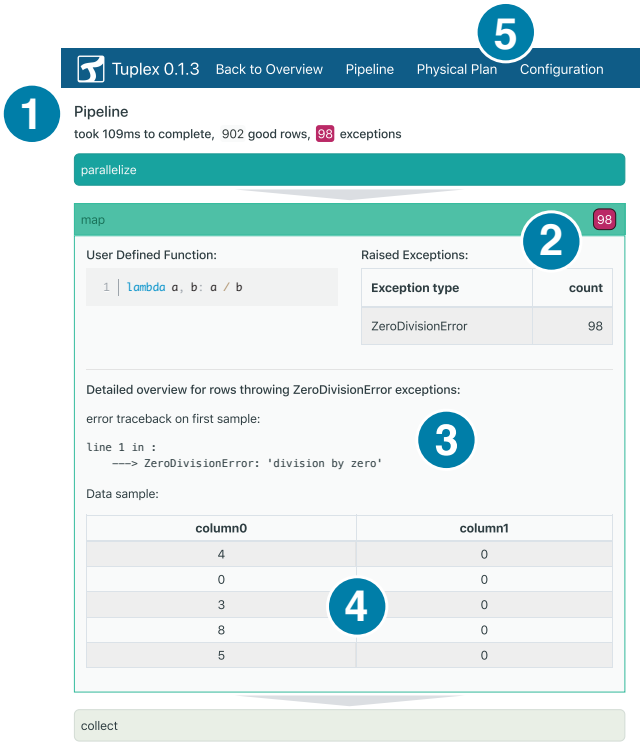


Figure 4: Rows causing errors are clustered by exception type and are saved for later resolution. For each operator and exception type, a sample of input data that caused such errors is displayed to the user via the *Tuplex WebUI*.

5. ACKNOWLEDGEMENTS

This research is funded in part by NSF Career Award IIS-1453171, Air Force YIP Award FA9550-15-1-0144, a Paris Kanellakis Graduate Fellowship and supported by Google, Intel, and Microsoft as part of the MIT DataSystems and AI Lab (DSAIL).

6. REFERENCES

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394. ACM, 2015.

[2] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An architecture for compiling udf-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.

[3] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. Mison: a fast json parser for data analytics. *PVLDB*, 10(10):1118–1129, 2017.

[5] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[6] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[7] G. Piatetsky. Python eats away at r: Top software for analytics, data science, machine learning in 2018: Trends and analysis, May 2018.

[8] picloud. The cloudpickle package. (acc. 11/25/2018).

[9] A. Rubin. Column store database benchmarks: Mariadb columnstore vs. clickhouse vs. apache spark - percona database performance blog. `https://www.percona.com/blog/2017/03/17/column-store-database-benchmarks\-mariadb-columnstore-vs-clickhouse-\vs-apache-spark/`, mar 2017. (acc. 03/18/2019).

[10] M. Stonebraker. Technical perspective - one size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12):76, 2008.

[11] G. Van Rossum et al. Python programming language. In *USENIX Annual Technical Conference*, volume 41, page 36, 2007.

[12] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *ACM SIGPLAN Notices*, volume 52, pages 662–676. ACM, 2017.

[13] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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*, pages 2–2. USENIX Association, 2012.