Efficient Execution of User-Defined Functions in SQL Queries

Yannis Foufoulas
University of Athens, Athena R.C.
Athens, Greece
johnfouf@di.uoa.gr

Alkis Simitsis
Athena Research Center
Athens, Greece
alkis@athenarc.gr

ABSTRACT

User-defined functions (UDFs) have been widely used to overcome the expressivity limitations of SQL and complement its declarative nature with functional capabilities. UDFs are particularly useful in today's applications that involve complex data analytics and machine learning algorithms and logic. However, UDFs pose significant performance challenges in query processing and optimization, largely due to the mismatch of the UDF execution and SQL processing environments. In this tutorial, we present state-of-the-art methods and systems towards efficient execution of UDFs in SQL queries. We focus on low-level techniques for physical optimization and compilation of UDF queries, describe and compare the core, recent approaches in the area, discuss their advantages and limitations, identify critical gaps in theory and practice, and propose promising future research directions.

PVLDB Reference Format:
Yannis Foufoulas and Alkis Simitsis. Efficient Execution of User-Defined Functions in SQL Queries. PVLDB, 16(12): 3874 - 3877, 2023.
doi:10.14778/3611540.3611574

1 INTRODUCTION

Relational databases, based on decades-old research and experience, provide many hooks for processing and managing efficiently large data volumes, offering features such as optimized (distributed) query processing, efficient storage, ACID properties, consistency, fault tolerance, and many others. Still, the SQL language provides limited expressive power, which cannot capture the data processing requirements that nowadays are routinely met in modern applications in data science, data analytics, edge computing, etc. To that end, all popular data engines support user-defined functions (UDFs) that extend the relational paradigm with syntactic and semantic support to capture complicated tasks and algorithms. However, the performance of executing queries with UDFs (i.e., UDF queries) inside a data engine is routinely subpar and creates significant bottlenecks largely due to the impedance mismatch between relational (SQL) evaluation and procedural (e.g., C/C++, Scala, Map-Reduce, Java, R, Matlab, Python) execution.

The early research efforts to improve performance of UDF queries focused on algebraic-style optimization and cost modeling of UDFs, with a special interest to Map-Reduce UDFs. Techniques considered for boosting UDF query performance included static partition and parallel execution, query rewriting and reordering for aggregate functions, sharing of partial aggregates, table UDF parallelization, and so on. Several of these efforts influenced commercial products such as Teradata, SQL Server, Apache Flink, etc. Moreover, such results offered developers a decent toolkit to handle UDFs designed for alternative implementation of relational operators (e.g., parallel join), schema transformations (e.g., one-to-many mappings, data lineage, generate/remove fields, split a record across multiple tables, derive new data from existing values), and data cleansing transformations (e.g., duplicate detection and removal, expressing data quality rules via integrity constraint checks).

In recent years, we experience an emerging and increasingly growing interest in more advanced UDF functionality emanating from applications in data science and data analytics, including machine learning (ML) pipelines, advanced data analytics (e.g., predictive and prescriptive analytics, text/video analytics), new and complex UDF types (e.g., analytic functions, ML algorithms and models, ELT and continuous load functions), etc. This trend has exacerbated significantly the UDF query performance problem and has led to a different class of solutions that focus on low-level techniques for physical optimization and compilation of UDF queries with an emphasis on UDFs coded in C/C++, Java, and Python. Python UDFs are particularly interesting as they (a) tend to be very popular among the growing communities of data science and data analytics [31], and (b) present intriguing and limiting performance challenges due to the conversions required between Python and C/C++, which is the implementation choice of most data engines. Presenting and comparing this new class of state-of-the-art solutions towards efficient execution of UDFs in SQL queries is the topic of this tutorial.

Tutorial scope, duration, and outline. The tutorial focuses on the problem of efficient execution of UDF queries from a systems perspective and present a comprehensive study answering questions such as: (a) why my UDF queries are slow, (b) how does the UDF execution landscape look like, (c) what have we learnt about optimizing UDF query performance and what is still missing, (d) what solution matches my application, and (e) how mature the current solutions are and what is their potential to impact systems in production. We propose a 90-min tutorial structured as follows:
(1) Introduction, challenges, and a taxonomy of solutions [∼10']
(2) UDF translation into SQL [∼20']
(3) UDF translation into an IR [∼20']
(4) UDF integration with data engines [∼30']
(5) Open issues and research directions [∼10']

Related surveys and tutorials. Rheinländer et al. [35] present a survey on optimizing UDF dataflows focusing on three core aspects: (a) syntactical dataflow modification, including variable and function inlining, group-by simplification, and query unnesting; (b) semantics inference and rewriting options for UDFs via annotations and code analysis; and (c) dataflow transformations toward redundancy elimination, partial aggregation, operator (de-)composition,

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611574

migration, and implementation. A first part of our tutorial was given at ICDE 2023 [12]. The tutorial presented a broad coverage of the approaches to UDF design and execution covering the early works on algebraic, logical optimization of UDFs in relational and object databases, as well as in data pipelines (such as Map-Reduce pipelines), and also an overview of the modern approaches to physical optimization. Following the very positive feedback and suggestions we received, we designed this tutorial to present in depth the more recent, state-of-the-art work on low-level UDF optimization and compilation, perform a multi-dimensional comparison of existing works to identify the current gaps and limitations, and conclude with a call to arms presenting remaining challenges and open problems in this area.

Targeted audience and learning output. The tutorial targets researchers and practitioners who are keen to know (a) the state-of-the-art practices and approaches to UDF query optimization and execution; (b) the technical limitations and the trade-offs between design choices and achieved goals; and (c) the new challenges and opportunities for data processing in modern data engines. This is an interdisciplinary tutorial comprising cutting-edge aspects from database systems and compilers research. Still no prior knowledge is needed on systems or compilers research, but we assume basic understanding of database and software concepts. The tutorial will be example-driven showcasing the strengths and limitations of the state of the art. The tutorial material will become publicly available.

2 THE UDF LANDSCAPE

Challenges. Several challenges render the optimization of UDF execution a non-trivial problem.

Fragmented space. Databases support UDFs in many languages, such as C/C++, Java, R, Matlab, Python, Scala, etc. And each language presents its own intricacies as in turn they support many libraries and frameworks. Hence, a one-size-fits-all solution does not seem as a straightforward solution.

Expressiveness and usability. Modern applications require flexibility in UDF definitions, namely, expressiveness features as variety of UDF types (e.g., scalar, aggregate, table, analytical, window), dynamically typed UDFs, stateful execution, as well as usability features such as parametric polymorphic UDFs and functional syntax.

Performance challenges. As UDFs typically run in an execution environment different than the data engine, there are significant overheads due to frequent context switches, data conversions and copies, potential materialization of intermediate results, excessive function calls, inefficient compilation, long UDF pipelines, and so on. Such issues also relate to the engine's execution model, i.e., iterator (Volcano) or operator/vector at a time, or data centric models.

Query Optimization. Query optimizers generally treat UDFs as a black-box, as they are not exposed to the UDF semantics and internal implementation. Techniques such as introspection [19] and code analysis help in enabling logical style optimizations like operator re-ordering or UDF push-down. Additional low-level techniques (e.g., operator fusion) also seem promising.

Classification of solutions. Motivated by such challenges, we classify the proposed approaches as follows. We start with the method of UDF integration with the data engine, either via UDF translation to SQL or to an internal representation (IR), or with engine-level UDF compilation and integration. Then, most approaches support UDFs in a specific programming language (with Python being a popular choice). We also consider the techniques supported organized as follows: (a) UDF optimization, (b) execution model, (c) query optimization, and (d) usability and expressiveness.

UDF optimization techniques include: parallelization, vectorization, function inlining, in-process or out-process execution, method or tracing just-in-time (JIT) compilation. The execution model of the data engine running the UDFs relates to how it processes the data (e.g., tuple/vector/operator at a time) and its layout as column or row store. Typical query optimization techniques enabled by the UDF approaches include: operator reordering and fusion, and rule or cost based heuristics. Another interesting dimension for the classification includes whether the techniques proposed are engine or library/framework specific and whether they support static or dynamic data types. An abridged schematic classification of the state of the art is shown in Figure 1. The tutorial will also cover additional dimensions in detail such as UDF types supported etc.

3 UDF SOLUTIONS

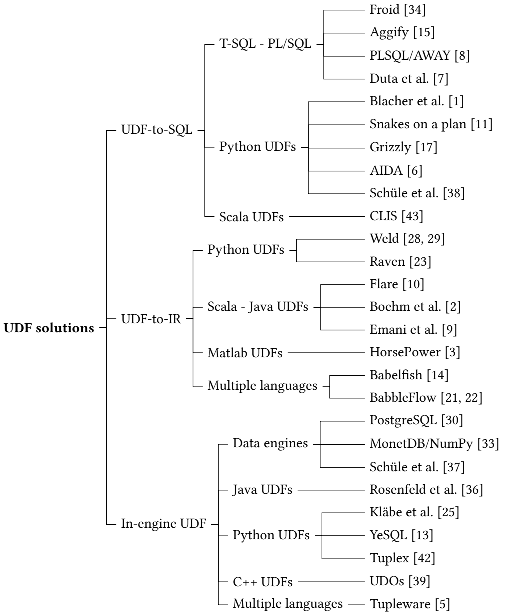
UDF translation into SQL. Several approaches translate UDFs written in various languages to semantically equivalent SQL [e.g., 4, 6–8, 17, 20, 34, 37, 40]. In general, these works propose general purpose optimizations, e.g., compilation optimizations and inlining, to reduce context switches between SQL and UDF.

Froid [34] (offered with SQL Server) rewrites loop-less T-SQL scalar UDFs into SQL and integrates them in the SQL query during binding, employing optimizations as dynamic slicing, constant folding, dead code elimination, and parallelization. Aggify [15] extends this logic to UDFs with cursor loops (loops over the query results) and rewrites them into SQL. PLSQL/AWAY [8] transforms PL/SQL functions with iterations into SQL queries using a recursive common table expression (CTE) `WITH RECURSIVE`. Follow-up work investigates efficient implementations of recursive CTEs using functional-style UDFs [7]. CLIS [43] optimizes Spark UDFs using lazy inductive synthesis to generate a sequence of decompositions that correspond to increasingly harder inductive synthesis problems.

A considerable volume of work focus on translating Python UDFs to SQL. Blacher et al. [1] translates Python variables, functions, conditions, loops, and errors, using mostly SQL's `WITH` clause and employing dynamic tuple-wise parallelization and pipelined SQL optimization. Grizzly [17] translates Pandas operations into SQL queries with Python UDFs. AIDA [6] provides abstractions for in-database analytics with Python UDFs and translates mainly linear algebra operation into MonetDB Numpy UDFs.

UDF translation into an IR. Another direction is to convert UDFs into an Intermediate Representation (IR) and then to SQL, which offers several optimizations and abstractions at the cost of being limited to specific libraries (e.g., Matlab, NumPy).

Weld [28, 29] optimizes computations across functions and libraries using a common IR (WeldIR). Weld focuses on data movement optimizations for data-parallel operators (e.g., relational, linear algebra), which tend to be time-consuming. HorsePower [3] rewrites Matlab UDFs into an array-based IR (HorseIR) using compiler optimization strategies to produce efficient machine code.



			UDF optimization							Execution model					Qry optimization				Usability & expr.			
			parallelization	vectorization	func inlining	in-process	out-process	tracing JIT	method JIT	tuple-at-a-time	vector-at-a-time	oper-at-a-time	column-based	row-based	reordering	fusion	rules	cost model	engine spec	lib spec	static	dynamic
UDF-to-SQL	T-SQL - PL/SQL	Froid [34]	x			x				x				x	x	x	x	x			x	
		Aggify [15]	x			x				x				x	x	x	x	x			x	
		PLSQL/AWAY [8]	x			x				x					x	x	x	x	x		x	
		Duta et al. [7]	x			x				x						x		x			x	
	Python UDFs	Blacher et al. [1]			x					x	x	x	x	x	x		x	x		x		x
		Snakes on a plan [11]			x					x	x	x	x	x	x		x	x		x		x
		Grizzly [17]			x					x	x	x	x	x	x		x	x		x	x	
		AIDA [6]	x		x							x	x				x					
		Schüle et al. [38]		x	x				x	x				x				x	x	x	x	
	Scala UDFs	CLIS [43]	x			x										x						
UDF-to-IR	Python UDFs	Weld [28, 29]	x	x		x						x	x			x			x	x	x	
		Raven [23]	x			x				x				x	x	x	x	x	x	x	x	
	Scala - Java UDFs	Flare [10]		x	x				x	x			x					x	x	x	x	
		Boehm et al. [2]		x	x						x		x			x	x	x	x	x	x	
		Emani et al. [9]			x					x				x				x	x	x	x	
	Matlab UDFs	HorsePower [3]	x		x	x						x	x			x	x		x	x		
	Multiple languages	Babelfish [14]	x			x			x	x						x	x		x			
		BabbleFlow [21, 22]	x		x	x				x		x		x	x	x	x	x	x		x	
In-engine UDF	Data engines	PostgreSQL [30]	x				x			x				x				x	x		x	
		MonetDB/NumPy [33]	x	x		x						x	x		x		x		x		x	
		Schüle et al. [37]	x			x				x				x	x	x	x	x	x		x	
	Java UDFs	Rosenfeld et al. [36]	x			x				x				x	x		x	x	x		x	
	Python UDFs	Kläbe et al. [25]	x	x			x				x		x				x				x	
		YeSQL [13]	x		x	x	x	x		x	x	x	x	x	x	x	x	x			x	x
		Tuplex [42]	x		x	x		x		x				x		x	x		x			x
	C++ UDFs	UDOs [39]	x			x				x				x							x	
	Multiple languages	Tupleware [5]	x		x	x		x		x				x	x	x	x	x	x		x	

Figure 1: An abridged classification of approaches to UDF execution in data engines

Raven [23] integrates ML runtimes with SQL Server for inference of trained ML models and employs an IR to enable cross-optimizations between ML and database operators. [9] extracts SQL from imperative code using a DAG based IR (D-IR) for applications, which is translated first to a functional representation (fold IR) and then into SQL. Flare [10] analyzes Spark's optimized plans to encode relational operators, data structures, Scala UDFs, data layout, and other configurations, in order to generate native code for execution. SystemML uses cost-based optimization of operator fusion plans over DAGs of linear algebra operations to generate Java code [2].

Other approaches deal with UDF pipelines written in multiple languages. BabbleFlow [21, 22] uses a unified representation (xLM [41]) to translate a hybrid flow expressed in various languages (e.g., SQL, Java, Javascript, Pig) to a semantically equivalent hybrid flow expressed in the same or a different set of languages through optimizations such as operator fusion. Babelfish [14] converts polyglot queries written in Java, Javascript, and Python to an IR (Babelfish IR). It performs IR-based operator fusion between built-in operators and UDFs in three steps: specialization, inlining, and scalar replacement.

UDF integration with data engines. Most data engines offer in-engine support for UDFs [e.g., 27, 30, 32]. Several research approaches have dealt with low-level, in-engine optimization of UDF queries, including (adaptive) compilation with JIT/LLVM [26] and vectorized execution [24]. Direct embedding of UDFs in native query execution engines has also been explored, e.g., in Spark [36].

Kläbe et al. [25] explores out-process execution of Python UDFs in Actian Vector with various compilation frameworks (e.g., Cython, Nuitka, Numba), vectorization, and multi-process parallelization.

Running UDFs in-process with the data engine eliminates overheads such as data exchange between processes. For example, MonetDB/NumPy [33] exploits vectorization and same data representation in MonetDB and Python, to avoid data conversions in a UDF call. UDO [39] integrates C++ user-defined operators, compiled in shared libraries, into existing query plans, retaining ACID properties. Schüle et al. [37] extends PostgreSQL's JIT compiler using LLVM to inline lambda expressions in table functions. YeSQL [13] goes a step further to operate with either server or embedded data engines, employing a tuple-at-a-time model for Python UDFs, providing a SQL extension with language features to enhance usability, and performing optimizations such as tracing JIT, stateful UDFs, parallelization, and operator fusion. Fusion works at a higher-level too; e.g., GOLAP [18] modifies pipelines to eliminate overheads derived from (de-)serialization steps. Mutable [16] decouples UDF optimization from low-level optimizations and delegates JIT compilation, optimization, and adaptive execution to an underlying engine (Google's V8) using WebAssembly as an IR.

A special category includes systems that integrate engine capabilities within the compilation, using LLVM as an IR (these systems could be classified as UDF-to-IR as well). For example, Tupleware [5] introspects UDFs and blends high-level query optimization with low-level LLVM compilation to provide a language-agnostic frontend for map-reduce style operators toward the automatic compilation of UDF analytical workflows. In the same spirit, Tuplex [42] is an end-to-end JIT compiler with LLVM for dynamically typed Python UDFs, offering compilation and code generation optimizations, and simple, logical query optimizations.

4 ISSUES AND OPPORTUNITIES (ABRIDGED)

Open issues. *Complexity.* SQL is not designed for multi-lingual data analytics, hence expressing complex algorithms in SQL usually results in cumbersome queries. *Fragmentation.* Most solutions involving translation to IR or SQL are tailored to specific UDF languages, frameworks, and libraries. *Performance.* UDF compilation and translation strategies introduce significant overheads, especially for short running queries. *Expressiveness.* Support for dynamically typed and stateful UDFs is not trivial.

Research Opportunities. *Modern pipelines involve UDFs.* Data engines should provide native support for various UDF types without translation overhead and treat dynamically typed UDFs as first class citizens. *Optimization.* Apply adaptive, multi-lingual fusion techniques for relational and procedural operations alike, without being limited to specific libraries/languages. *Compilation.* Catch-up with recent advancements in related areas; for example instead of re-engineering compilers, consider employing emerging compilers (e.g., PyPy) and techniques (e.g., WebAssembly - WASM [16]).

5 PRESENTERS

Yannis Foufoulas received his PhD degree from University of Athens in 2023, and currently, he is a research associate at Athena Research Center. He works in the EU-funded projects OpenAIRE-Nexus and Human Brain Project and his research interests include modern databases, query optimization, and data/text in-database analytics.

Alkis Simitsis is a Research Director at Athena Research Center. Previously, he held positions with HP/HPE Labs, Micro Focus, Unravel Data, and IBM Research, including Chief/Principal Scientist. He has 20+ years of experience building innovative solutions for scalable big data infrastructure, data-intensive analytics, distributed databases, and systems optimization. He holds 45 patents, has published 110+ papers (7000+ citations, h-index: 43), and frequently serves in various roles in PC's of top-tier int'l scientific conferences.

REFERENCES

[1] Mark Blacher, Joachim Giesen, Sören Laue, Julien Klaus, and Viktor Leis. 2022. Machine Learning, Linear Algebra, and More: Is SQL All You Need?. In *CIDR.*

[2] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *PVLDB* 11, 12 (2018).

[3] Hanfeng Chen, Joseph Vinish D'silva, Laurie J. Hendren, and Bettina Kemme. 2021. HorsePower: Accelerating Database Queries for Advanced Data Analytics. In *EDBT.* 361–366.

[4] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *SIGPLAN.* 3–14.

[5] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. 2015. An Architecture for Compiling UDF-centric Workflows. *PVLDB* 8, 12 (2015), 1466–1477.

[6] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. 2018. AIDA - Abstraction for Advanced In-Database Analytics. *PVLDB* 11, 11 (2018).

[7] Christian Duta and Torsten Grust. 2020. Functional-Style SQL UDFs With a Capital 'F'. In *SIGMOD.* 1273–1287.

[8] Christian Duta, Denis Hirn, and Torsten Grust. 2020. Compiling PL/SQL Away. In *CIDR.*

[9] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. 2016. Extracting Equivalent SQL from Imperative Code in Database Applications. In *SIGMOD.* ACM, 1781–1796.

[10] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *USENIX.*

[11] Tim Fischer, Denis Hirn, and Torsten Grust. 2022. Snakes on a Plan: Compiling Python Functions into Plain SQL Queries. In *SIGMOD.* ACM, 2389–2392.

[12] Yannis E. Foufoulas and Alkis Simitsis. 2023. User-Defined Functions in Modern Data Engines. In *ICDE.* IEEE.

[13] Yannis E. Foufoulas, Alkis Simitsis, Eleftherios Stamatogiannakis, and Yannis E. Ioannidis. 2022. YeSQL: "You extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases. *PVLDB* 15, 10 (2022).

[14] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. 2022. Babelfish: Efficient Execution of Polyglot Queries. *PVLDB* 15, 2 (2022), 196–210.

[15] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *SIGMOD.*

[16] Immanuel Haffner and Jens Dittrich. 2023. A simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries. In *EDBT.* 1–13.

[17] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting Pandas in a Box. In *CIDR.*

[18] Anna Herlihy, Periklis Chrysogelos, and Anastasia Ailamaki. 2022. Boosting Efficiency of External Pipelines by Blurring ApplicationBoundaries. In *CIDR.*

[19] Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. 2013. Peeking into the optimization of data flow programs with MapReduce-style UDFs. In *ICDE.*

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

[21] Petar Jovanovic, Alkis Simitsis, and Kevin Wilkinson. 2014. BabbleFlow: a translator for analytic data flow programs. In *SIGMOD.* 713–716.

[22] Petar Jovanovic, Alkis Simitsis, and Kevin Wilkinson. 2014. Engine independence for logical analytic flows. In *ICDE.* 1060–1071.

[23] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *CIDR.*

[24] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *PVLDB* 11, 13 (2018).

[25] Steffen Kläbe, Robert DeSantis, Stefan Hagedorn, and Kai-Uwe Sattler. 2022. Accelerating Python UDFs in Vectorized Query Execution. In *CIDR.*

[26] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *IEEE/ACM CGO.* 75–88.

[27] MonetDB. 2022. User Defined Functions. Available at: https://www.monetdb.org/documentation-Sep2022/dev-guide/sql-extensions/user-defined-functions.

[28] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. In *PVLDB.*

[29] Shoumik Palkar, James Thomas, Anil Shanbhag, Malte Schwarzkopf, Saman P. Amarasinghe, and Matei Zaharia. 2017. A Common Runtime for High Performance Data Analysis. In *CIDR.*

[30] PostgreSQL. 2022. PL/pgSQL, SQL Procedural Language. Available at: https://www.postgresql.org/docs/current/plpgsql.html.

[31] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Matteo Interlandi, Avrilia Floratou, Konstantinos Karanasos, Wentao Wu, Ce Zhang, Subru Krishnan, Carlo Curino, and Markus Weimer. 2019. Data Science through the looking glass and what we found there. *CoRR* abs/1912.09536 (2019). http://arxiv.org/abs/1912.09536

[32] PySpark. 2022. Available at: https://pypi.org/project/pyspark.

[33] Mark Raasveldt and Hannes Mühleisen. 2016. Vectorized UDFs in Column-Stores. In *SSDBM.* 16:1–16:12.

[34] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB* 11, 4 (2017), 432–444.

[35] Astrid Rheinländer, Ulf Leser, and Goetz Graefe. 2017. Optimization of Complex Dataflows with User-Defined Functions. *ACM Comput. Surv.* 50, 3 (2017).

[36] Viktor Rosenfeld, René Müller, Pinar Tözün, and Fatma Özcan. 2017. Processing Java UDFs in a C++ environment. In *SoCC.* 419–431.

[37] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. 2020. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *SSDBM.* 6:1–6:12.

[38] Maximilian E. Schüle, Luca Scalerandi, Alfons Kemper, and Thomas Neumann. 2023. Blue Elephants Inspecting Pandas: Inspection and Execution of Machine Learning Pipelines in SQL. In *EDBT.* OpenProceedings.org, 40–52.

[39] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *PVLDB* 15, 5 (2022).

[40] Varun Simhadri, Karthik Ramachandra, Arun Chaitanya, Ravindra Guravannavar, and S. Sudarshan. 2014. Decorrelation of user defined function invocations in queries. In *ICDE.* 532–543.

[41] Alkis Simitsis and Kevin Wilkinson. 2014. The specification for xLM: an encoding for analytic flows. Technical Report, HP Labs.

[42] Leonhard F Spiegelberg, Rahul Yesantharao, Malt Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *SIGMOD.*

[43] Guoqiang Zhang, Yuanchao Xu, Xipeng Shen, and Isil Dillig. 2021. UDF to SQL translation through compositional lazy inductive synthesis. *OOPSLA* 5 (2021).