User-Defined Functions in Modern Data Engines

Yannis Foufoulas
Univ. of Athens and Athena Research Center, Greece
johnfouf@di.uoa.gr

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

***Abstract*—Modern data management applications involve complex processing tasks over large volumes of data. Although this falls naturally within the scope of relational databases, many such tasks cannot be expressed in SQL and require additional expressive power achieved via user-defined functions (UDFs). However, efficient processing of UDFs in data engines hinge on dealing with the impedance mismatch between UDF execution and SQL processing. In recent years, the problem of efficient UDF execution in modern data engines has gained significant traction. In this tutorial, we present recent advancements in this area, involving a broad scope of solutions ranging from algebraic, cost-based optimization to low level, physical query optimization, compilation, and execution. We also describe limitations and open issues, and discuss promising future research directions.**

I. Introduction and Motivation

The diversity and complexity of modern data management applications accentuate the limitations of SQL and have led to SQL extensions with syntactic and semantic support for user-defined functions (UDFs) [e.g., 1, 46, 47, 50, 53, 70]. Early research approaches focused on providing data engines with additional computational capability by enabling functional-style programming alongside with declarative queries. This functionality however comes with a significant performance overhead due to the impedance mismatch between UDF evaluation and relational processing. More recent efforts have shifted the focus on performance optimization and proposed techniques inspired by systems and compilers research to boost UDF execution in modern data engines.

Typical data analytics and data science applications employ SQL queries containing a combination of standard relational operations and multiple calls to UDFs, often in a pipelined fashion. Although this enables expressing more complex logic with SQL queries, several challenges significantly affect query performance. A holistic approach to efficiently integrate UDFs with SQL should address challenges such as context switching, data conversion and data copies, materialization of intermediate results, performance boosters as JIT-compiled execution, opening up UDFs to query optimization, and so on.

In this tutorial, we review techniques, algorithms, and systems towards efficiently integrating UDFs with SQL, including approaches to logical and physical optimization of UDF queries, i.e., SQL queries containing UDFs. Early approaches dealt with challenges such as algebraic-style optimization and cost modeling of UDFs, with a special interest to Map-Reduce UDFs. In recent years, the research community has shown an emerging and increasingly growing interest in approaches emanating from needs in areas such as data analytics and data science, focusing on physical level query optimization and compilation with an emphasis on UDFs coded in C/C++, Java, and Python. Python UDFs are particularly interesting as they (a) tend to be the most popular amongst the growing communities of data science and data analytics [54], 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.

II. Tutorial Scope and Coverage

We study the execution of UDF queries from a systems perspective and aim at answering questions such as: (a) how far along have we went so far with optimizing UDF queries, what have we learnt and what is still missing, (b) whether low level, query compilation provides us with a competitive advantage over or along with traditional query optimization, and (c) whether low level, fine-tuned UDF query optimization and execution has the potential to impact systems in production.

Duration and Outline. We propose a 90-min tutorial:
(1) Introduction and a taxonomy of solutions [∼10']
(2) Logical, algebraic optimization of UDF queries [∼15']
(3) Integrating UDFs with data engines [∼20']
(4) Translating UDFs into SQL [∼20']
(5) Translating UDFs into IR [∼15']
(6) Open issues and research directions [∼10']

Our presentation will cover the works listed next, with analytical examples and multidimensional comparisons of their offerings and limitations (Fig. 1 shows an abridged example).

A. A Taxonomy of Solutions

The related work can be classified based on the following dimensions: (a) UDF characteristics, (b) execution environment characteristics, and (c) solution characteristics.

The UDF characteristics include: (i) UDF categories: SQL, procedural, multilingual; (ii) UDF programming model: functional, map-reduce; (iii) UDF type: scalar, aggregate, table, other; (iv) UDF optimization: parallelization, vectorization, function inlining, in/out-process, JIT compilation; and (v) UDF integration: UDF in engine, UDF to SQL, UDF to IR. The execution environment characteristics include: (i) Execution model: tuple-at-a-time, vector-at-a-time, operator-at-a–time, column or row based; (ii) Query optimization: operator reordering, operator fusion, rules, cost model; and (iii) Data type support: static, dynamic. Finally, additional information of interest include properties of the solution proposed, such as: (i) Pluggable solution: whether it is tied to a specific

	UDF integration			UDF optimization						UDF categories			Execution model					Query optimization				Pluggable				Data types	
	UDF in engine	UDF to SQL	UDF to IR	parallelization	vectorization	function inlining	in/out - process	tracing JIT	method JIT	SQL UDFs	Procedural	Multilingual	tuple-at-a-time	vector-at-a-time	oper-at-a-time	column-based	row-based	reordering	fusion	rules	cost model	engine specific	lib specific	UDF language	interpreter (I), compiler (C), transpiler (T)	static	dynamic
MonetDB [56]	x			x	x		in			x	x				x	x		x		x		x		SQL, C, Python	I, C	x	
Postgres [53]	x			x			out			x	x		x				x				x	x		SQL, C, Python	I, C	x	
PySpark [55]	x			x	x		out				x		x				x	x		x	x	x		Python	I	x	
Tuplex [69]	x			x		x	in	x			x		x				x		x	x		x		Python	C		x
Tuppleware [12]	x			x		x	in	x			x		x				x	x	x	x	x	x		LLVM	T to IR	x	
UDOs [65]	x			x			in				x		x				x							C++	C	x	
YeSQL [19]	x			x		x	both	x			x		x	x	x	x	x	x	x	x	x			Python	JIT C	x	x
[42]	x			x	x		in				x			x		x				x				Python	C, T	x	
[62]	x			x			in				x		x				x	x		x	x	x		Java	C	x	
[64]	x			x			in				x		x				x	x	x	x	x	x		C	T to IR	x	
CLIS [73]		x		x			in			x									x					SQL	T to SQL		
Froid [57]		x		x			in			x			x				x	x	x	x	x			SQL	T to SQL	x	
PLSQL/Away [16]		x		x			in			x			x					x	x	x	x	x		SQL	T to SQL	x	
[15]		x		x			in			x			x						x		x			SQL	T to SQL	x	
HorsePower [9]			x	x		x	in				x				x	x			x	x		x	x	MATLAB	T to IR		
Raven [40]			x	x			in				x		x				x	x	x	x	x	x	x	Python	T to IR	x	
Weld [51]			x	x	x		in				x				x	x			x			x	x	Python	T to IR	x	
Babelfish [23]			x	x			in		x			x	x						x	x		x		Python, JS	JIT C		
BabbleFlow [39]			x	x		x	in					x	x		x		x	x	x	x	x	x		SQL,Java,JS,Pig	I	x	

Fig. 1. An abridged taxonomy of approaches to integrating UDFs with data engines

architecture or can be applied to various environments (e.g., engine specific, library/package specific, programming language specific); (ii) Usability: available artifacts, reproducible, open-source; and (iii) Evaluation setup: datasets, workloads.

An abridged taxonomy of modern research approaches and popular system offerings is illustrated in Figure 1.

B. Early Approaches to Optimizing UDF Queries

Early works focused on the logical optimization of UDF queries based on rules, templates, annotations, etc. These works can be classified based on the programming model they adopt, namely functional programming and map-reduce.

1) Optimization of UDF queries: The placement of expensive predicates and scalar UDFs within an execution plan first studied in the early 90's [7, 29, 30]. [48] shows that finding optimal orderings of predicates and functions (map operators) under factorization is NP-hard. Furthermore, UDF optimization has been studied in the context of Object RDBMS [6, 27, 28, 34, 35]. [45] describes data type and function extensions (UDFs in PASCAL) in HDBL, a SQL-based query language for complex objects. [6] presents an optimization framework for UDFs focusing on runtime execution strategies based on optimizer statistics. [34, 35] employ a static partition strategy to enable optimization and parallel execution of aggregate and table UDFs. [27] proposes a self-tuning cost model that adapts to changing UDF execution patterns by leveraging a dynamic quadtree-based summary structure and considers prediction accuracy, average prediction cost, and average model update costs. [28] extends this idea to self-tuning cost models based on an instance-based technique using conventional k-nearest neighbor (KNN) with an R*-tree. [11] uses query optimization and rewriting to boost user-defined aggregate functions (UDAFs) by extending transformations such as eager group-by, eager count, and double eager. SUDAF [71, 72] describes recent results on rewriting partial aggregations of UDAFs.

2) Optimization of map-reduce style UDFs: [33] argues that a set of properties, rather than a full algebraic specification, suffice to establish reordering conditions for data processing operators. Such properties can be accurately estimated for black-box operators by statically analyzing the general-purpose code of their user-defined functions. Sudo [74] analyzes UDFs and reasons about functional and data-partition properties in map-reduce computations, in order to explore optimization opportunities for data shuffling. Sofa [60] a rule-based, logical optimizer for map-reduce UDFs employs a set of rewrite templates and an operator-property graph to capture relationships between operators and properties, such as algebraic properties, operator cost model, operator throughput, operator parallelization (e.g., map, reduce), etc. It uses Meteor, a declarative dataflow language [59].

There is also work towards parallelizing UDF execution. SQL/MapReduce [20] (Aster Data's SQL/MR) performs table UDF parallelization designed dynamically based on the context during query optimization in a distributed relational database. That is, the output schema of a UDF is specified by the function at query plan-time (polymorphic functions). [22] uses UDF annotations to enable optimizations for parallel execution of relational operators and map-reduce UDFs. These annotations include pre-conditions (e.g., partitioning bounds, local grouping, sorting), post-conditions (e.g., sorting, grouping, generated values, order preserving), and optimizer hints (e.g., deterministic function, output size, expected runtime).

[61] surveys techniques for optimizing dataflows with UDFs, including syntactical dataflow modification (e.g., variable and function inlining, group-by simplification, query unnesting), inferring semantics and rewrite options for UDFs (e.g., annotations, code analysis), and dataflow transformations (e.g., operator (de-)composition, redundancy elimination, operator migration, partial aggregation, operator implementation).

C. Modern Approaches to Optimizing UDF Queries

Recent approaches have focused mainly on the physical optimization of UDF queries. Efficient UDF execution has been approached in three directions: (a) integrating UDFs with the data engine, (b) translating UDF code into SQL, and (c) translating UDF code into an internal representation (IR).

1) UDF integration with the data engine: Besides in-engine support for UDFs in popular systems [e.g., 47, 53, 55], several research approaches have dealt with low-level, in-engine optimization of UDF queries, including compiling queries with LLVM, adaptive compilation, and vectorized execution.

In one line of work, the data is processed in-process in the data engine, hence, eliminating data import/export between processes. As an example, UDO [65] integrates user-defined operators, compiled in shared libraries, into existing query plans, and also retains ACID properties during UDF execution. [56] integrates Python/NumPy with MonetDB, exploiting vectorized execution and using the same representation of the underlying data in both MonetDB and Python, thus avoiding unnecessary conversions of data values in a UDF call.

Another research direction exploits various compilation optimizations to execute UDFs efficiently. [42] explores techniques to execute Python UDFs in a DBMS (e.g., Actian Vector), and in particular: (a) compilation frameworks (i.e., Cython [2], Nuitka [49], Numba [43]), (b) vectorized UDF execution [41], (c) parallelization via multiprocessing, and compares these against interpreted CPython UDFs. It considers out-of-process UDF execution and employs socket connections to achieve process communication with the database. YeSQL [19] works with either server or embedded data engines and employs a tracing JIT, tuple-at-a-time model for Python UDF execution. It provides an extension to SQL supporting language features that enhance usability, and optimizations such as tracing JIT, stateful UDFs, parallelization, and UDF fusion. Fusion could also work at a higher-level, e.g., modify external pipelines to eliminate overheads derived from (de-)serialization steps as in GOLAP [31]. Tupleware [12] employs UDF workflow compilation using code generation optimization heuristics based on properties of the data, UDFs, and underlying hardware. It blends high-level query optimization with low-level compiler techniques, using LLVM [44] to provide a language-agnostic front-end for map-reduce style operators, introspect UDFs [5, 32], and enable low-level optimizations. Tuplex [69] presents an optimized, end-to-end JIT compiler for Python, with performance that outperforms several popular data engines. [64] enriches PostgreSQL with lambda expressions and subqueries as table function's arguments, and also extends PostgreSQL's JIT compiler using LLVM to inline lambda expressions in table functions.

Mutable [25] decouples UDF optimization from low-level optimizations and delegates JIT compilation, optimization, and adaptive execution to an underlying engine. It uses WebAssembly as an IR and Google's V8 as backend and generates specialized algorithms and data structures directly in WebAssembly. This enables holistic optimization without the drawback of long code generation and compilation time.

Direct embedding of UDFs in native query execution engines has also been explored. As an example, [62] explores direct UDF embedding in Spark. At UDF registration, its bytecode is injected into the embedded JVM. At query time, the instance of the generated UDF and the data from the engine's data buffers are also passed to the embedded JVM.

2) Translating UDFs into SQL: Several works translate UDFs written in various languages to semantically equivalent SQL [e.g., 10, 13, 15, 16, 26, 36, 58, 64, 66]. Collectively these approaches capture a rich set of UDF types and propose general purpose optimizations to minimize context switches between the declarative SQL and the imperative UDF, such as compilation optimizations and inlining.

Froid [58] (MS SQL Server's Scalar UDF Inlining) rewrites loop-less T-SQL UDFs into SQL and integrates them in the SQL query during binding. It focuses on scalar UDFs and at compilation, it uses optimizations such as dynamic slicing, constant folding, dead code elimination, and parallelization. BlackMagic [57] demonstrates Froid's functionality for inlining scalar UDFs into SQL queries. Aggify [24] extends this logic to UDFs with cursor loops (loops over the query results) and rewrites them into SQL (not as an optimization feature).

PLSQL/AWAY [16] transforms PL/SQL functions containing iterations into regular SQL queries by employing a recursive common table expression (CTE) `WITH RECURSIVE`; essentially, it compiles iterations into SQL-level recursion. Context switches can be reduced by executing the newly created queries along with their embracing queries. [15] investigates efficient implementations of recursive CTEs using functional-style UDFs. [14] argues that recursive UDFs implemented as functional-style UDFs seem to be a good choice for expressing CTEs from a developer's point of view as well (e.g., less error-prone, easier to understand).

CLIS [73] optimizes Spark SQL programs containing Scala UDFs. Based on the observation that most practical UDF-to-SQL translation tasks are amenable to a fine-grained decomposition strategy, CLIS employs a lazy inductive synthesis approach that generates a sequence of decompositions that correspond to increasingly harder inductive synthesis problems.

Finally, a considerable volume of work relates to Python UDFs. For example, [3] translates Python into SQL showing how algorithmic primitives of procedural languages can be mapped to PostgreSQL's declarative syntax. It translates variables, functions, conditions, loops, and errors, using mostly SQL' `WITH` clause. It also shows that techniques such as dynamic tuple-wise parallelization and pipelined SQL optimization are superior to NumPy that executes a query in separate parts. Another line of work introduces abstractions to transparently define and execute Python UDFs in data processing systems. Grizzly [26] is a front-end framework that exposes a Pandas interface and translates Pandas operations into SQL queries with Python UDFs. AIDA [13] targets linear algebra and provides abstractions for in-database analytics with Python UDFs. It exposes interfaces similar to popular Python statistic packages and translates them into MonetDB Numpy UDFs to support fast linear algebra operations.

3) Translating UDFs into an IR: Another direction is to convert UDFs into an Intermediate Representation (IR), which can then be holistically translated into SQL and run in a data engine. In general, IR-based approaches come at the cost of supporting specific libraries (e.g., Matlab, NumPy), which should be rewritten in the IR.

Weld [51, 52] 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 [9] rewrites a query into an array-based IR (HorseIR [8]) and apply compiler optimization strategies developed for array-based languages to produce efficient machine code for execution. HorsePower supports MATLAB UDFs. *Raven* [40] is a prototype system that integrates ML runtimes (e.g., ONNX Runtime) with SQL Server and employs an IR to enable cross-optimizations between ML and database operators. Raven's prime goal is inference of already trained ML models. [17] extracts SQL from imperative code using a DAG based IR (D-IR) for applications, which is translated first to a functional representation called fold IR (F-IR) and then into SQL. [4] integrates in SystemML an approach to cost-based optimization of operator fusion plans over DAGs of linear algebra operations, which finally are translated to Java code. [18] Flare analyzes Spark's optimized plan to construct a computation graph that encodes relational operators, data structures, Scala UDFs, data layout, and other configurations. It finally generates native code for execution.

There are also approaches that involve UDFs written in more than one language and serve as multi-language translators. Example systems include BabbleFlow and Babelfish. BabbleFlow [38, 39, 68] translates a hybrid flow expressed in different languages (e.g., SQL, Java, Javascript, Apache Pig) to a semantically equivalent hybrid flow expressed in the same or a different set of languages. To this end, it translates a hybrid flow into a unified flow language called xLM [67], enabling several graph transformations (e.g., (de-)composition) and optimization. Babelfish [23] 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. It uses Truffle's [37] partial evaluation to specialize operator implementations according to runtime parameters. Then, it relies on Graal's [21] inlining heuristics to inline calls to external libraries within UDF, create a unified instruction graph, and thus, achieve operator fusion. Finally, it uses scalar replacement to remove intermediate objects.

Note, that works compiling queries into a low-level form (e.g., LLVM, JIT) could also be classified under this category as LLVM or JIT could be seen as a form of IR.

III. Open issues and Research Directions

Open issues. Pain points in terms of expressiveness and performance: (i) Statically typed UDFs does not fit well with high-level programming languages; (ii) Stateful UDFs burdens the UDF developer with the responsibility to handle shared state across parallel UDFs; (iii) SQL was not initially designed for multilingual data analytics through UDFs, thus expressing complex algorithms in SQL usually results in cumbersome queries; (iv) Current approaches (e.g., IRs) are often limited to specific UDF languages or specific libraries; (v) Optimization of UDF queries is not trivial, especially, for newly created UDFs that often operate as black boxes; and (vi) UDF compilation/translation strategies introduce a significant overhead, especially for short running queries.

Research Directions. There are several directions worth pursuing: (i) Extend SQL language to make UDFs first class citizens; (ii) Generalize an architecture for UDFs so that a data processing system would be able to process existing functional code. This would allow data scientists scale their ML modules without rewriting them to match the specific data processing system's design; (iii) Support fully dynamically typed UDFs, since in real scenarios data analytics usually run directly on messy data; (iv) Adaptively fuse relational execution with procedural evaluation to balance short and long running queries; (v) Apply multilingual fusion techniques to holistically optimize procedural UDF execution without targeting specific libraries; and (vi) Consider existing and compatible works on compilers instead of re-engineering compilers (e.g, JIT) for data management. For example, YeSQL and Mutable report efficient execution using PyPy and Wasm, respectively. (vii) Explore alternative execution models, e.g., containerized UDFs in local/remote settings [63].

IV. Target Audience and Learning Output

Material. The tutorial will be example-driven showcasing the strengths and limitations of the state of the art. The tutorial material will become publicly available.

Audience. It comprises students, researchers, and practitioners who want to understand the current state of the art and the future directions of UDF query optimization and execution. No prior knowledge is needed on systems or compilers research, but we assume basic understanding of database concepts.

Output. The learning output includes: (a) Understanding use cases and existing approaches to UDF query optimization and execution. (b) Understanding the technical limitations and the trade-offs between design choices and achieved goals. And (c) exposure to the new challenges and opportunities for data processing coming along with modern data engines.

V. Presenters

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

Alkis Simitsis is a Research Director at Athena Research Center. Previously, he held various positions with HP/HPE Labs, Micro Focus, Unravel Data, and IBM Research, including Chief/Principal Scientist. He has 18+ years of experience building innovative solutions for scalable big data infrastructure, data-intensive analytics, distributed databases, and systems optimization. He holds 43 patents, has published 110+ papers (6500+ citations, h-index: 43), and frequently serves in various roles in PC's of top-tier int'l scientific conferences.

Acknowledgements. This work has been partially supported by EU Horizon 2020 programmes INODE and HBP-SGA3 (grant agreement No 863410 and 945539, respectively).

REFERENCES

[1] Amazon Redshift, "Creating user-defined functions. Available at: https://docs.aws.amazon.com/redshift/latest/dg/user-defined-functions.html," 2022.

[2] S. Behnel, R. Bradshaw, C. Citro, L. Dalcín, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 31–39, 2011.

[3] M. Blacher, J. Giesen, S. Laue, J. Klaus, and V. Leis, "Machine learning, linear algebra, and more: Is sql all you need?" in *CIDR*, 2022.

[4] M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare, "On optimizing operator fusion plans for large-scale machine learning in systemml," *PVLDB*, vol. 11, no. 12, pp. 1755–1768, 2018.

[5] M. J. Cafarella and C. Ré, "Manimal: Relational optimization for data-intensive programs," in *WebDB*, 2010.

[6] F. Cariño and W. O'Connell, "Plan-per-tuple optimization solution - parallel execution of expensive user-defined functions," in *VLDB*, 1998, pp. 690–695.

[7] S. Chaudhuri and K. Shim, "Optimization of queries with user-defined predicates," *ACM Trans. Database Syst.*, vol. 24, no. 2, pp. 177–228, 1999.

[8] H. Chen, J. V. D'silva, H. Chen, B. Kemme, and L. Hendren, "HorseIR: Bringing array programming languages together with database query processing," in *ACM SIGPLAN DLS*, 2018, p. 37–49.

[9] H. Chen, J. V. D'silva, L. J. Hendren, and B. Kemme, "HorsePower: Accelerating database queries for advanced data analytics," in *EDBT*, 2021, pp. 361–366.

[10] A. Cheung, A. Solar-Lezama, and S. Madden, "Optimizing database-backed applications with query synthesis," in *SIGPLAN*, 2013, pp. 3–14.

[11] S. Cohen, "User-defined aggregate functions: bridging theory and practice," in *SIGMOD*, 2006, pp. 49–60.

[12] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik, "An architecture for compiling udf-centric workflows," *PVLDB*, vol. 8, no. 12, pp. 1466–1477, 2015.

[13] J. V. D'silva, F. De Moor, and B. Kemme, "AIDA - abstraction for advanced in-database analytics," *PVLDB*, vol. 11, no. 11, pp. 1400–1413, 2018.

[14] C. Duta, "Another way to implement complex computations: functional-style SQL UDF," in *HILDA w/ SIGMOD*. ACM, 2022, pp. 6:1–6:7.

[15] C. Duta and T. Grust, "Functional-Style SQL UDFs With a Capital 'F'," in *SIGMOD*, 2020, pp. 1273–1287.

[16] C. Duta, D. Hirn, and T. Grust, "Compiling PL/SQL away," in *CIDR*, 2020.

[17] K. V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan, "Extracting equivalent SQL from imperative code in database applications," in *SIGMOD*. ACM, 2016, pp. 1781–1796.

[18] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf, "Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data," in *USENIX*. USENIX Association, 2018, pp. 799–815.

[19] Y. E. Foufoulas, A. Simitsis, E. Stamatogiannakis, and Y. E. Ioannidis, "Yesql: "you extend sql" with rich and highly performant user-defined functions in relational databases," *PVLDB*, vol. 15, no. 10, pp. 2270–2283, 2022.

[20] E. Friedman, P. M. Pawlowski, and J. Cieslewicz, "Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions," *PVLDB*, vol. 2, no. 2, pp. 1402–1413, 2009.

[21] GraalVM, "Available at: https://www.graalvm.org," 2022.

[22] P. Große, N. May, and W. Lehner, "A study of partitioning and parallel UDF execution with the SAP HANA database," in *SSDBM*.

[23] P. M. Grulich, S. Zeuch, and V. Markl, "Babelfish: Efficient execution of polyglot queries," *PVLDB*, vol. 15, no. 2, pp. 196–210, 2022.

[24] S. Gupta, S. Purandare, and K. Ramachandra, "Aggify: Lifting the curse of cursor loops using custom aggregates," in *SIGMOD*, 2020, pp. 559–573.

[25] I. Haffner and J. Dittrich, "A simplified architecture for fast, adaptive compilation and execution of SQL queries," in *EDBT*, 2023, pp. 1–13.

[26] S. Hagedorn, S. Kläbe, and K. Sattler, "Putting pandas in a box," in *CIDR*, 2021.

[27] Z. He, B. S. Lee, and R. R. Snapp, "Self-tuning UDF cost modeling using the memory-limited quadtree," in *EDBT*, ser. Lecture Notes in Computer Science, vol. 2992. Springer, 2004, pp. 513–531.

[28] ——, "Self-tuning cost modeling of user-defined functions in an object-relational DBMS," *ACM Trans. Database Syst.*, vol. 30, no. 3, pp. 812–853, 2005.

[29] J. M. Hellerstein and J. F. Naughton, "Query execution techniques for caching expensive methods," in *SIGMOD*, 1996, pp. 423–434.

[30] J. M. Hellerstein and M. Stonebraker, "Predicate migration: Optimizing queries with expensive predicates," in *SIGMOD*, 1993, pp. 267–276.

[31] A. Herlihy, P. Chrysogelos, and A. Ailamaki, "Boosting efficiency of external pipelines by blurring applicationboundaries," in *CIDR*, 2022.

[32] F. Hueske, M. Peters, A. Krettek, M. Ringwald, K. Tzoumas, V. Markl, and J. Freytag, "Peeking into the optimization of data flow programs with mapreduce-style udfs," in *ICDE*, 2013, pp. 1292–1295.

[33] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas, "Opening the black boxes in data flow optimization," *PVLDB*, vol. 5, no. 11, pp. 1256–1267, 2012.

[34] M. Jaedicke and B. Mitschang, "On parallel processing of aggregate and scalar functions in object-relational DBMS," in *SIGMOD*, 1998, pp. 379–389.

[35] ——, "User-defined table operators: Enhancing extensibility for OR-DBMS," in *VLDB*, 1999, pp. 494–505.

[36] A. Jindal, K. V. Emani, M. Daum, O. Poppe, B. Haynes, A. Pavlenko, A. Gupta, K. Ramachandra, C. Curino, A. Mueller, W. Wu, and H. Patel, "Magpie: Python at speed and scale using cloud backends," in *CIDR*, 2021.

[37] Johannes Kepler University, "The truffle language implementation framework. available at: http://www.ssw.uni-linz.ac.at/Research/Projects/JVM/Truffle.html," 2022.

[38] P. Jovanovic, A. Simitsis, and K. Wilkinson, "Babbleflow: a translator for analytic data flow programs," in *SIGMOD*, 2014, pp. 713–716.

[39] ——, "Engine independence for logical analytic flows," in *ICDE*, 2014, pp. 1060–1071.

[40] K. Karanasos, M. Interlandi, F. Psallidas, R. Sen, K. Park, I. Popivanov, D. Xin, S. Nakandala, S. Krishnan, M. Weimer, Y. Yu, R. Ramakrishnan, and C. Curino, "Extending relational query processing with ML inference," in *CIDR*, 2020.

[41] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. A. Boncz, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *PVLDB*, vol. 11, no. 13, pp. 2209–2222, 2018.

[42] S. Kläbe, R. DeSantis, S. Hagedorn, and K.-U. Sattler, "Accelerating python udfs in vectorized query execution," in *CIDR*, 2022.

[43] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-Based Python JIT Compiler," in *LLVM-SC*, 2015.

[44] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *IEEE/ACM CGO*, 2004, pp. 75–88.

[45] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch, and M. Wallrath, "Design and implementation of an extensible database management system supporting user defined data types and functions," in *VLDB*, 1988, pp. 294–305.

[46] Microsoft, "Transact-SQL Reference. Available at: https://learn.microsoft.com/en-us/sql/t-sql," 2022.

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

[48] T. Neumann, S. Helmer, and G. Moerkotte, "On the optimal ordering of maps and selections under factorization," in *ICDE*, 2005, pp. 490–501.

[49] Nuitka the Python Compiler, "Available at: https://nuitka.net," 2022.

[50] Oracle, "PL/SQL. Available at: https://www.oracle.com/database/technologies/appdev/plsql.html," 2022.

[51] S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. P. Amarasinghe, S. Madden, and M. Zaharia, "Evaluating end-to-end optimization for data analytics applications in weld," *PVLDB*, vol. 11, no. 9, pp. 1002–1015, 2018.

[52] S. Palkar, J. Thomas, A. Shanbhag, M. Schwarzkopf, S. P. Amarasinghe, and M. Zaharia, "A common runtime for high performance data analysis," in *CIDR*, 2017.

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

[54] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, A. Floratou, K. Karanasos, W. Wu, C. Zhang, S. Krishnan, C. Curino, and M. Weimer, "Data science through the looking glass and what we found there," *CoRR*, vol. abs/1912.09536, 2019. [Online]. Available: http://arxiv.org/abs/1912.09536

[55] PySpark, "Available at: https://pypi.org/project/pyspark," 2022.

[56] M. Raasveldt and H. Mühleisen, "Vectorized udfs in column-stores," in *SSDBM*, 2016, pp. 16:1–16:12.

[57] K. Ramachandra and K. Park, "Blackmagic: Automatic inlining of scalar udfs into SQL queries with froid," *PVLDB*, vol. 12, no. 12, pp. 1810–1813, 2019.
[58] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. A. Galindo-Legaria, and C. Cunningham, "Froid: Optimization of imperative programs in a relational database," *PVLDB*, vol. 11, no. 4, pp. 432–444, 2017.
[59] A. Rheinländer, M. Beckmann, A. Kunkel, A. Heise, T. Stoltmann, and U. Leser, "Versatile optimization of udf-heavy data flows with sofa," in *SIGMOD*, 2014, pp. 685–688.
[60] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann, "SOFA: an extensible logical optimizer for udf-heavy data flows," *Inf. Syst.*, vol. 52, pp. 96–125, 2015.
[61] A. Rheinländer, U. Leser, and G. Graefe, "Optimization of complex dataflows with user-defined functions," *ACM Comput. Surv.*, vol. 50, no. 3, pp. 38:1–38:39, 2017.
[62] V. Rosenfeld, R. Müller, P. Tözün, and F. Özcan, "Processing java udfs in a C++ environment," in *SoCC*.
[63] K. Saur, T. Mirmira, K. Karanasos, and J. Camacho-Rodríguez, "Containerized execution of udfs: An experimental evaluation," *PVLDB*, vol. 15, no. 11, pp. 3158–3171, 2022.
[64] M. E. Schüle, J. Huber, A. Kemper, and T. Neumann, "Freedom for the sql-lambda: Just-in-time-compiling user-injected functions in postgresql," in *SSDBM*, 2020, pp. 6:1–6:12.
[65] M. Sichert and T. Neumann, "User-defined operators: Efficiently integrating custom algorithms into modern databases," *PVLDB*, vol. 15, no. 5, pp. 1119–1131, 2022.
[66] V. Simhadri, K. Ramachandra, A. Chaitanya, R. Guravannavar, and S. Sudarshan, "Decorrelation of user defined function invocations in queries," in *ICDE*, 2014, pp. 532–543.
[67] A. Simitsis and K. Wilkinson, "The specification for xLM: an encoding for analytic flows," Technical Report, HP Labs, 2014.
[68] A. Simitsis, K. Wilkinson, U. Dayal, and M. Hsu, "HFMS: managing the lifecycle and complexity of hybrid analytic data flows," in *ICDE*, 2013, pp. 1174–1185.
[69] L. F. Spiegelberg, R. Yesantharao, M. Schwarzkopf, and T. Kraska, "Tuplex: Data science in python at native code speed," in *SIGMOD*, 2021, pp. 1718–1731.
[70] Vertica, "Extending Vertica. Available at: https://www.vertica.com/docs/12.0.x/HTML/Content/Authoring/ExtendingVertica/ExtendingVertica.htm," 2022.
[71] C. Zhang and F. Toumani, "Sharing computations for user-defined aggregate functions," in *EDBT*, 2020, pp. 241–252.
[72] C. Zhang, F. Toumani, and B. Doreau, "SUDAF: sharing user-defined aggregate functions," in *ICDE*, 2020, pp. 1750–1553.
[73] G. Zhang, Y. Xu, X. Shen, and I. Dillig, "UDF to SQL translation through compositional lazy inductive synthesis," *OOPSLA*, vol. 5, pp. 1–26, 2021.
[74] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Li, W. Lin, J. Zhou, and L. Zhou, "Optimizing data shuffling in data-parallel computation by understanding user-defined functions," in *USENIX*, 2012, pp. 295–308.