QFusor: A UDF Optimizer Plugin for SQL Databases

Konstantinos Chasialis[1], Theoni Palaiologou[1,2], Yannis Foufoulas[1,2], Alkis Simitsis[1], Yannis Ioannidis[1,2]
[1]*Athena Research Center,* [2]*University of Athens*
Athens, Greece
kostasch@athenarc.gr, cs2210019@di.uoa.gr, johnfouf@di.uoa.gr, alkis@athenarc.gr, yannis@di.uoa.gr

***Abstract*—Modern data applications in areas such as text mining, document analysis, and data science, involve complex algorithms and logic that cannot be expressed in SQL. Therefore, SQL databases employ user-defined functions (UDFs) to extend their supported functionality. However, this comes at a significant performance cost as UDFs routinely become the bottleneck in query execution. To deal with this problem, we present QFusor, an optimizer plugin for UDF queries in relational databases. QFusor minimizes the performance overheads introduced by the impedance mismatch between the UDF and SQL execution environments by employing techniques such as vectorization, parallelization, tracing JIT compilation, and operator fusion for various types of UDF (scalar, aggregate, table UDFs) and relational operators. QFusor follows a pluggable, engine-agnostic design and can work with several popular SQL databases offering a significant boost in their UDF query performance.**

I. Introduction

Modern applications implement complex logic and algorithms that cannot be expressed as declarative SQL operations. Examples include complex schema changing and data cleansing transformations, deriving new data from existing values (e.g., compute the interval between a given moment and a timestamp withing the data), document analysis, text mining and analytics, etc. Therefore, SQL databases extend their query expressiveness with procedural functionality via user-defined functions (UDFs). UDFs is a powerful tool for usability, but unfortunately, they present a significant performance challenge. In SQL queries containing UDFs (UDF queries), the bottleneck will most likely be the UDF, irrespective of the complexity of the query or the size of the data [9].

The research solutions could be classified as follows [6, 5]: (a) converting UDFs into SQL [e.g., 3, 10, 13], (b) converting UDFs into an internal representation (IR) [e.g., 2, 4, 8, 11, 14, 16], and (c) in-database UDF integration [e.g., 7, 12, 15]. Translating UDFs to either SQL or IR comes with several benefits, such as straightforward query optimization (SQL) and advanced low level compilation (IR), but typically these works support only specific packages/libraries. Standalone processing engines (e.g., Tupleware [2], Tuplex [16]) execute end-to-end Python pipelines, but they do not consider UDF queries executed in a SQL database. Efforts in the latter category explore low-level optimization techniques, such as vectorization and JIT compilation, but usually to a limited extent; e.g., they consider fusion of only scalar UDFs.

In this paper, we demonstrate QFusor, an optimizer plugin for Python UDF queries in SQL databases. QFusor employs techniques such as parallelization, tracing just-in-time (JIT) compilation, and operator fusion for scalar, aggregate, table UDFs and relational operators. In doing so, QFusor eliminates unnecessary overheads in UDF compilation and execution, and hence, reduces excessive function calls and context switches (e.g., in tuple-at-a-time execution), minimizes data conversions and copies, and avoids materialization of intermediate results (e.g., in operator-at-a-time execution). It also opens up unseen query optimization opportunities. QFusor follows a pluggable, engine-agnostic design and works with several SQL databases such as SQLite, DuckDB, MonetDB, PostgreSQL, Greeplum, etc. significantly boosting their UDF query performance.

Our work supports OpenAIRE (openaire.eu), a technical infrastructure built and supported by a consortium of 65 European universities, research centers, and other institutions, which yearly services 45M requests and 1M visits. OpenAIRE connects to 1000+ data providers, and to date has harvested over 130M publications, 2M datasets, 85K research software artifacts, and 1.5M research projects from 23 funders. It uses 150+ UDFs to perform text mining, information extraction, and text analytics over Open Access publications. The QFusor demonstration utilizes production queries from OpenAire and showcases through it the effectiveness of our techniques to boost UDF queries in various SQL databases.

II. Demonstrable Features

A. Demo Scenario #1

Our first query (see Figure 1) implements an analytics use-case on publication and funding data, which assesses how research projects affect the collaboration among scientists in terms of papers they co-authored. The data is collected from heterogeneous sources, such as OpenAIRE, PubMed, arXiv, crossref, Zenodo, Cordis, etc., and needs to be cleansed and homogenized. The query identifies all author pair combinations per publication. For each project, it computes the number of author pairs that collaborated in this project and co-authored at least one paper during the project, the number of author pair collaborations before the project start, and the number of collaborations after the project end.

The query contains several cleansing functions to (a) avoid false negatives, (b) normalize authors names ($lower$) and remove short terms (e.g., middle names), (c) sort the authors, (d) flatten author pairs to multiple rows ($combinations$), (e) retrieve funding information (funder's name, funding class, project id), and (f) normalize dates into a common format ($cleandate$). Finally, it retrieves all publications authored by each author pair and aggregates the result by funder,

```
WITH pairs(pubid, pubdate, projectstart, projectend, funder, class,
  project, authorpair) AS (
  SELECT pubid,pubdate, projectstart, projectend,
    extractid(project) AS projectid,
    extractfunder(project) AS funder,
    extractclass(project) AS class,
    combinations(jsort(jsortvalues(
      removeshortterms(lower(authors)))), 2) AS authorpair
  FROM pubs
)
SELECT  funder, class, projectid,
  SUM(CASE WHEN cleandate(pubdate) between projectstart and projectend
      THEN 1 ELSE NULL END) AS authors_during,
  SUM(CASE WHEN cleandate(pubdate) < projectstart
      THEN 1 ELSE NULL END) AS authors_before,
  SUM(CASE WHEN cleandate(pubdate) > projectend
      THEN 1 ELSE NULL END) AS authors_after
FROM (
    SELECT  projectpairs.funder, projectpairs.class, projectpairs.projectid,
            pairs.pubdate, projectpairs.projectstart, projectpairs.projectend,
            pairs.authorpair
    FROM (
           SELECT * FROM pairs WHERE projectid IS NOT NULL
         ) AS projectpairs, pairs
    WHERE projectpairs.authorpair = pairs.authorpair
)
GROUP BY funder, class, projectid;
```

Fig. 1. Example UDF query

```
WITH pairs(pubid,pubdate, projectstart, projectend, funder, class, project,
   authorpair) AS (
   SELECT * FROM FUSED_UDF1((SELECT * FROM pubs))
)
SELECT * FROM FUSED_UDF2((
    select  projectpairs.funder, projectpairs.class, projectpairs.project,
            pairs.pubdate, projectpairs.projectstart, projectpairs.projectend,
            pairs.authorpair
    from (select * from pairs where project is not null) AS projectpairs, pairs
    where projectpairs.authorpair = pairs.authorpair
));
```

Fig. 2. Example fused UDF query

funding class, and project. This query offers fusion opportunities as most of its UDFs are candidates for fusion: $extractid$, $extractfunder$, $extractclass$, $lower$, $jsortvalues$, $jsorted$, $removeshortterms$, $combinations$. It also presents an opportunity of fusing functions (*cleandate*, *sum*) and relational operators (i.e., $filters$, *case*); filters and *case* can be JIT implemented as Python UDFs and run in a single, fused UDF. Figure 2 shows a fused version of the query.

B. System Overview

Figure 3 shows an overview of QFusor pluggable architecture. We consider two user roles (orange in the figure): (a) the *application user* who submits a query, and (b) the *UDF developer* who creates and registers UDFs to the database. Currently, QFusor supports scalar, aggregate, and table Python UDFs.

QFusor comprises two phases (blue in the figure), an offline that involves UDF registration to the database UDF manager, and an online that operates at query time and includes the following steps: (a) query admission, (b) discovery of fusible operators, (c) fusion optimization, (d) fused code generation, and (e) query rewrite. One of QFusor unique features is that it seamlessly connects as a plugin to a SQL database (gray color in the figure), either embedded (e.g., SQLite, DuckDB) or server-based (e.g., MonetDB, PostgreSQL). It only requires that the database offers: a plan generation mechanism (a query optimizer), a UDF registration mechanism, and support for C UDFs. Additional databases can be added.

QFusor optimizes the relational and procedural parts of a query in an holistic manner. It employs a stateful mechanism to enable fusion of UDFs or UDFs and relational operators in SQL databases. Operator fusion combines multiple operators into one, while pipeline operators could be further placed into the same hot-loop. Beside reducing excessive function calls, context switches, and data conversion/copy,

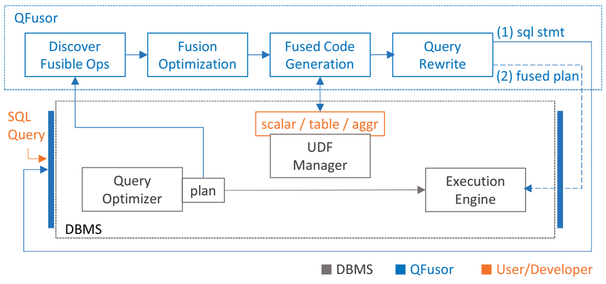


Fig. 3. Pluggable system architecture

this also provides the UDF JIT compiler with larger chunks of code enabling more holistic optimization in the UDFs execution itself. Moreover, QFusor introduces a new query optimization step (see Section II-C) that considers various UDF properties (types, data dependencies, execution statistics) and alters the execution plan enabling fusion opportunities for UDF and SQL operators.

Our demonstration starts with showcasing a step-by-step UDF query execution, using *Demo Scenario #1* as an example.

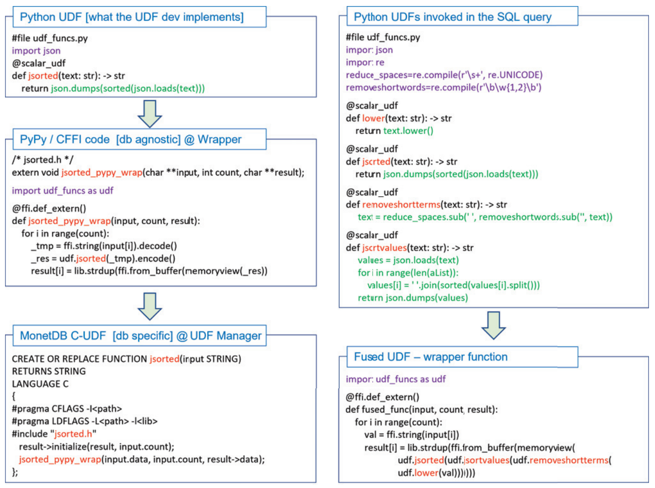
C. Demonstrating the Optimization Pipeline

1) UDF registration: The UDF registration mechanism automatically wraps, embeds, and registers to the database either a new UDF (offline operation) or an online JIT-created fused UDF (occurs at query runtime). Figure 4-left shows the registration of $jsorted$. The mechanism requires as metadata the UDF type (scalar, aggregate, table) via a decorator and its output data type(s). QFusor creates a wrapper function (middle figure) that converts the input data to C data objects, includes a call the Python UDF, and assigns the results to a C data object that is accessible to the database. Next, QFusor JIT-compiles the wrapper function into dynamically loadable objects that are embedded into a C UDF, and registers this C UDF to the database with a `CREATE FUNCTION` statement (bottom figure).

2) Query admission: At query time, the user issues a SQL query (Figure 1). QFusor adds a thin client layer to the database (thin blue line left and right of the DBMS in Figure 3) that parses the input SQL query looking for UDFs. If the query contains UDFs, the client probes the query optimizer (via `EXPLAIN`) and propagates the query plan produced to QFusor. Queries without UDFs are processed as usual.

3) Discovery of fusible operators: Next, we show how QFusor looks for fusion opportunities. To that end, it parses the query plan to discover *fusible operators*, i.e., operators that have a flow data dependency to each other (the output of the first is propagated to the input of the second). Two or more consecutive fusible operators in a plan form a *fusible section*. There are two cases for potential operator fusion: (c1) $udf \rightarrow udf$; and (c2) $udf \rightarrow rel$ (or $rel \rightarrow udf$).

Inspired by the compiler theory, we formulate the problem of discovering data dependencies among the operators of the plan as a *dependence analysis* problem and build a data flow graph (DFG) representing such dependencies. Then, the task at hand is to identify patterns of the (c1) to (c2) cases on the DFG. QFusor uses dynamic programming (DP) to recursively navigate DFG until no more fusible operators can be found.


Python UDF [what the UDF dev implements]

```
#file udf_funcs.py
import json
@scalar_udf
def jsorted(text: str): -> str
   return json.dumps(sorted(json.loads(text)))
```

PyPy / CFFI code [db agnostic] @ Wrapper

```
/* jsorted.h */
extern void jsorted_pypy_wrap(char **input, int count, char **result);

import udf_funcs as udf

@ffi.def_extern()
def jsorted_pypy_wrap(input, count, result):
   for i in range(count):
      _tmp = ffi.string(input[i]).decode()
      _res = udf.jsorted(_tmp).encode()
      result[i] = lib.strdup(ffi.from_buffer(memoryview(_res))
```

```
CREATE OR REPLACE FUNCTION jsorted(input STRING)
RETURNS STRING
LANGUAGE C
{
#pragma CFLAGS -I<path>
#pragma LDFLAGS -L<path> -l<lib>
#include "jsorted.h"
  result->initialize(result, input.count);
  jsorted_pypy_wrap(input.data, input.count, result->data);
};
```

```
#file udf_funcs.py
import json
import re
reduce_spaces=re.compile(r'\s+', re.UNICODE)
removeshortwords=re.compile(r'\b\w{1,2}\b')

@scalar_udf
def lower(text: str): -> str
   return text.lower()

@scalar_udf
def jsorted(text: str): -> str
   return json.dumps(sorted(json.loads(text)))

@scalar_udf
def removeshortterms(text: str): -> str
    text = reduce_spaces.sub(' ', removeshortwords.sub('', text))

@scalar_udf
def jsortvalues(text: str): -> str
    values = json.loads(text)
    for i in range(len(aList)):
       values[i] = ' '.join(sorted(values[i].split()))
    return json.dumps(values)
```

```
import udf_funcs as udf

@ffi.def_extern()
def fused_func(input, count, result):
   for i in range(count):
      val = ffi.string(input[i])
      result[i] = lib.strdup(ffi.from_buffer(memoryview(
                  udf.jsorted(udf.jsortvalues(udf.removeshortterms(
                  udf.lower(val))))))
```

Fig. 4. UDF registration and UDF fusion examples

The result of this step is a set of fusible sections. Figure 5 shows the fusion steps for the example UDF query in GraphViz generated by QFusor; colors indicate fusible sections.

4) Fusion optimization: Next, QFusor determines what operators should be fused together. It follows a hybrid, cost-based and rule-based approach to minimize the overheads occurring at the parts of the query where relational evaluation in SQL interacts with procedural evaluation of a UDF.

The UDF cost comprises the *UDF processing cost* and the *UDF wrapping cost* capturing the overhead of the data copy, transformation, and materialization involved. The processing cost depends on various execution parameters. But, as the wrapper functionality is concrete and measurable, QFusor tries to minimize (at least) the excessive wrapping cost. Therefore, for the (c1) case, QFusor always recommends fusion to save at a minimum the wrapping cost and also provide the tracing JIT compiler with larger traces to enable better optimization in the Python execution. We will demonstrate this with the running example; an abridged snippet is shown in Figure 4-right.

For the (c2) case that involves procedural UDFs and relational operations, QFusor considers two options. The first is to run the relational operator in the same execution environment as the UDF. This way, it is treated as a (c1) case, avoiding intermediate result materialization, enabling fusion optimization strategies such as loop fusion, etc. On the other hand, these gains may be optimized out by the presumably faster implementation of the operator in the database, which is the second option. The decision relies on two factors: (a) the relational operator's complexity and (b) its selectivity obtained by database stats. More complex, performance-sensitive operators shall preferably run in the data engine. Lower selectivity operators (returning many values) are good candidates for execution within a UDF; as the more data the operator processes, the larger the fusion benefit is due to the eliminations of data copy and transformation. To avoid a cold start for newly registered UDFs, QFusor uses a set of heuristics based on common logic and extensive experimentation. At steady phase, QFusor employs a Bayesian optimization learning technique inspired by [1] to collect UDF statistics and model UDF cost.

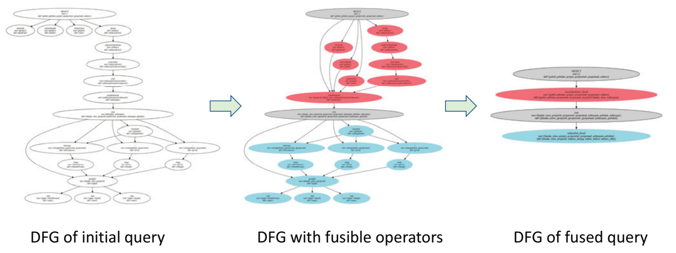


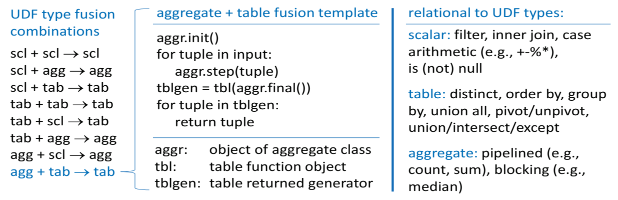
Fig. 5. Example UDF query fusion

Blending UDFs with relational operators may remove optimization barriers in a query plan when plan-wide query optimization is blocked by black-box treatment of UDFs (e.g., due to unknown semantics or stats). QFusor considers a commutative property: $udf_1 \rightarrow rel \rightarrow udf_2$ may be reordered as $rel \rightarrow udf_1 \rightarrow udf_2$, currently, in the conservative scenario that rel and udf_1 do not operate on the same attribute(s). Other policies work too [e.g., 17]. Operator reordering is a function of the query optimizer. We do not anticipate to replace this. We just enable additional fusion opportunities. If for example the fusion of the two UDFs is blocked by rel in the middle, reordering creates a data dependency between the two UDFs. Similarly, with fusion, we may enable additional query optimization opportunities as well; e.g., push down a relational operator or push-up an expensive UDF, which otherwise would not be considered by a query optimizer blocked by UDFs.

In Demo Scenario #1, the join and $cleandate$ have data dependency and as this join increases the data size, reordering these two is a potential option. Our demonstration will showcase several optimization cases based on real-world queries.

5) Fused code generation: QFusor generates just-in-time a Python source file that comprises a wrapper function implementing the fused operators, using techniques as *function inlining* and *loop fusion*. Then, QFusor registers the wrapper function to the database as we described earlier. When the code is JIT compiled, the tracing JIT compiler (currently, PyPy) automatically inlines and vectorizes function calls before locating hot traces for JIT compilation hence, avoiding unnecessary function call overheads. The wrapper function runs in the same process with the database, thus, eliminating any inter-process communication overhead between the database and the UDFs. Note that the wrapper function calls the UDFs in a try-except block, to handle gracefully possible execution errors in the UDFs. Figure 4-right shows the fusion of four UDFs of the running example: $jsorted$, $jsortvalues$, $removeshortterms$, and $lower$, and the wrapper function automatically produced.

Figure 6-left shows the possible fusion combinations involving scalar (scl), aggregate (agg), and table (tbl) UDFs. QFusor code generator chooses a function template to produce the fused code. For example, the fusion of aggregate and table UDFs produce a table UDF using the code template shown in the figure (middle). Figure 6-right shows a mapping of relational operators to UDF types (scalar, aggregate, or table) based on their inputs and outputs; once converted to UDFs then the UDF fusion templates can be used for relational operators as well. We devise two mechanisms to treat relational operators


UDF type fusion combinations

scl + scl → scl
scl + agg → agg
scl + tab → tab
tab + tab → tab
tab + scl → tab
tab + agg → agg
agg + scl → agg
agg + tab → tab

aggregate + table fusion template

```
aggr.init()
for tuple in input:
    aggr.step(tuple)
tblgen = tbl(aggr.final())
for tuple in tblgen:
    return tuple
```

aggr:	object of aggregate class
tbl:	table function object
tblgen:	table returned generator

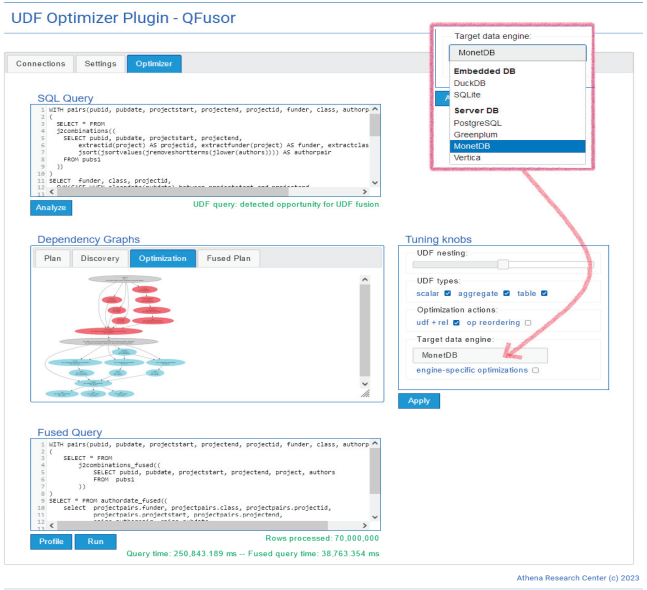
relational to UDF types:

scalar: filter, inner join, case arithmetic (e.g., +-%*), is (not) null

table: distinct, order by, group by, union all, pivot/unpivot, union/intersect/except

aggregate: pipelined (e.g., count, sum), blocking (e.g., median)

Fig. 6. UDF and relational fusion templates



```
WITH pairs(pubid, pubdate, projectstart, projectend, projectid, funder, class, authorp
(
  SELECT * FROM
  j2combinations((
    SELECT pubid, pubdate, projectstart, projectend,
        extractid(project) AS projectid, extractfunder(project) AS funder, extractclas
        jsort(jsortvalues(jremoveshortterms(jlower(authors)))) AS authorpair
    FROM pubs1
  ))
)
SELECT  funder, class, projectid,
```

UDF query: detected opportunity for UDF fusion

Target data engine:
MonetDB
Embedded DB
DuckDB
SQLite
Server DB
PostgreSQL
Greenplum
MonetDB
Vertica

UDF nesting:

UDF types:
scalar [✓] aggregate [✓] table [✓]

Optimization actions:
udf + rel [✓] op reordering [ ]

Target data engine:
MonetDB
engine-specific optimizations [ ]

```
WITH pairs(pubid, pubdate, projectstart, projectend, projectid, funder, class, authorp
(
    SELECT * FROM
        j2combinations_fused((
            SELECT pubid, pubdate, projectstart, projectend, project, authors
            FROM  pubs1
        ))
)
SELECT * FROM authordate_fused((
    select  projectpairs.funder, projectpairs.class, projectpairs.projectid,
            projectpairs.projectstart, projectpairs.projectend,
```

Rows processed: 70,000,000
Query time: 250,843.189 ms -- Fused query time: 38,763.354 ms

Fig. 7. A snapshot of QFusor GUI for our example query

as procedural UDFs: (a) implement them in Python, and (b) call the database source code (for systems allowing it).

An example such a system is MonetDB. QFusor leverages the C foreign function interface (CFFI) for Python, which enables the interaction of Python code with external C code, here, with MonetDB's shared libraries. Exporting MonetDB's internal functions enables the execution of group-by from a Python UDF with no additional overhead. The group-by still benefits from the database optimizations and features (e.g., indices, cache). We connect to the database through raw pointers in the engine's memory and access its stack through the exported functions. In other words, neither the data nor the functions on it leave the database. The same process can be applied for all relational operations (e.g. join, order-by). We will demonstrate this process for the *group by* of the example.

6) Query rewrite: Finally, QFusor rewrites the SQL query by replacing the affected operators in the query with the newly produced fused UDF(s). Then, QFusor reissues a rewritten SQL statement to the database (see Figure 3, path 1), which in turn produces a new query plan and dispatch it for execution; hence, in our current implementation, it does not undergo another QFusor process. As an alternative path, for MonetDB, QFusor may also produce a valid execution plan (MAL code) that can be sent directly to the execution engine (path 2).

III. Our Presentation

Presentation. Our presentation script starts with a step-by-step optimization process using the running example –Demo scenario #1– on the supported database engines (on average, fusion makes this query 5.3x faster). Then, we continue with other real-world queries on the OpenAIRE graph, such as:

Demo scenario #2: Compute the Jaccard similarity on document abstracts from different repositories.

Demo scenario #3: Compute the TF/IDF for open access publications fulltexts.

Demo scenario #4: A DML query for metadata enrichment: pre-process metadata from external sources and update the open science graph (e.g., add funding for published research).

The next phase of our presentation involves 10 canned queries over the zillow and flights datasets to show a live comparison with representative systems from the families of UDF-to-IR: Weld [11], Tuplex [16]; UDF-to-SQL: Grizzly [10]; and UDF-in-engine: YeSQL[7] and UDO [15].

User interaction. For off-script presentation, the audience will be able to run ad hoc queries with existing UDFs or new UDFs (should someone creates one lively). The interaction will be either via a terminal (which supports autocomplete, help, history, and more) or via the QFusor GUI (see Figure 7).

Acknowledgments. This work has been partially supported by the EU-HORIZON programmes FAIR-CORE4EOSC (GA.101057264), CREXDATA (GA.101092749), EBRAINS 2.0 (GA.101147319), and H2020-MSCAITN-2020 DEDS (GA.955895).

References

[1] O. Alipourfard et al., "CherryPick: Adaptively unearthing the best cloud configurations for big data analytics," in *USENIX NSDI*, 2017.
[2] A. Crotty et al., "An architecture for compiling udf-centric workflows," *PVLDB*, 2015.
[3] C. Duta, D. Hirn, and T. Grust, "Compiling PL/SQL away," in *CIDR*, 2020.
[4] K. V. Emani et al., "Extracting equivalent SQL from imperative code in database applications," in *SIGMOD*, 2016.
[5] Y. Foufoulas and A. Simitsis, "Efficient execution of user-defined functions in SQL queries," *PVLDB*, 2023.
[6] Y. Foufoulas and A. Simitsis, "User-defined functions in modern data engines," in *ICDE*, 2023.
[7] Y. E. Foufoulas et al., "YeSQL: "you extend SQL" with rich and highly performant user-defined functions in relational databases," *PVLDB*, 2022.
[8] P. M. Grulich, S. Zeuch, and V. Markl, "Babelfish: Efficient execution of polyglot queries," *PVLDB*, 2021.
[9] S. Gupta and K. Ramachandra, "Procedural extensions of SQL: understanding their usage in the wild," *PVLDB*, 2021.
[10] S. Hagedorn, S. Kläbe, and K. Sattler, "Putting pandas in a box," in *CIDR*, 2021.
[11] S. Palkar et al., "Evaluating end-to-end optimization for data analytics applications in weld," in *PVLDB*, 2018.
[12] M. Raasveld and H. Mühleisen, "Vectorized udfs in column-stores," in *SSDBM*, 2016.
[13] K. Ramachandra et al., "Froid: Optimization of imperative programs in a relational database," *PVLDB*, 2017.
[14] H. Shahrokhi et al., "Efficient query processing in python using compilation," in *SIGMOD*, 2023.
[15] M. Sichert and T. Neumann, "User-defined operators: Efficiently integrating custom algorithms into modern databases," *PVLDB*, 2022.
[16] L. F. Spiegelberg et al., "Tuplex: Data science in python at native code speed," in *SIGMOD*, 2021.
[17] C. Yan, Y. Lin, and Y. He, "Predicate pushdown for data science pipelines," in *SIGMOD*, 2023.