# Optimizing UDF Queries in SQL Data Engines

Konstantinos Chasialis*
Athena Research Center, EPFL
Athens, Greece
kostasch@athenarc.gr

Yannis Foufoulas
Athena Research Center
Athens, Greece
johnfouf@athenarc.gr

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

Yannis Ioannidis
Univ. of Athens, Athena R.C.
Athens, Greece
yannis@di.uoa.gr

## Abstract

Many SQL database engines support user-defined functions (UDFs) for complex in-database computations. However, UDFs often suffer from high execution overhead and limited optimization, as query planners treat them as black boxes. In this paper, we introduce QFusor, a pluggable optimizer that enhances UDF performance in SQL databases by: (a) reducing overhead through operator fusion, inlining, and stateful, JIT-compiled execution, (b) fusing different UDF types (scalar, aggregate, table) and blending them with relational operators, (c) exposing new optimization opportunities, and (d) enabling longer JIT compilation traces. QFusor integrates with various SQL databases and delivers up to 40x speedup over current research and commercial systems.

## Keywords

Query optimization, UDFs, fusion, JIT compilation, stateful.

## 1 Introduction

SQL databases have been the backbone of data analysis for over four decades, offering key advantages like query optimization, robust execution, scalable storage, and ACID guarantees. However, one of their core strengths, the declarative, set-oriented SQL language, also limits their ability to express complex analytical logic. To overcome this, modern databases support user-defined functions (UDFs), allowing developers to implement custom logic in languages like C/C++, Java, or Python [e.g., 3, 48, 51, 56, 60, 85]. While UDFs enhance expressiveness and usability, they often suffer from a performance mismatch with relational processing. In queries with UDFs, the bottleneck will most likely be the UDF, irrespective of the complexity of the calling query or the size of the data [33].

The performance challenges are exacerbated in queries with many UDFs. A common pattern in data science applications is the use of *long pipelines* composed of multiple UDFs [54, 55]. Data scientists often break down complex algorithms into multiple, modular UDFs for better reusability and productivity. However, this design limits performance even further as modern databases excel at fast, vectorized execution, and isolated UDFs running in separate contexts prevents aggressive compiler optimizations and underutilizes database capabilities.

In this paper, we deal with the following problem: "*How to optimize SQL queries containing UDFs (a.k.a. UDF queries) and execute them efficiently in a SQL database engine*". Toward this direction, we present QFusor, a stateful approach to optimizing UDF performance in SQL databases by blending together UDF operators (currently, Python UDFs) and/or UDF and relational operators, potentially enabling additional query optimizations, at a very low execution cost through stateful and just-in-time (JIT) execution.

*Work done while with Athena Research Center.

*EDBT '26, Tampere (Finland)*


*JIT execution.* JIT compilation boosts performance of a program by compiling parts of it to machine code at runtime. In contrast to method-based JIT compilers that translate one method at a time, tracing JIT can be more efficient as it uses frequently executed loops –hot loops– as the unit of compilation [e.g., 27, 71, 79]. This concept fits perfectly to UDFs, as they typically execute frequent calculations iteratively through the tuples of a table. Still, in order to fully exploit the benefits of tracing JIT, one needs to feed it with long traces –i.e., longer sequences of instructions.

*Operator fusion.* We enable this with *operator fusion*. Operator fusion combines multiple operators into one, and in the same loop, hence eliminating context switches, materialization of intermediate results (when applicable), data conversions and copies between the data engine and the UDF's execution environment, and also provides the UDF tracing JIT compiler with larger chunks of code enabling more holistic optimization of UDF execution, especially for long UDF pipelines, and thus, fully exploiting its tracing feature.

*Query optimization.* Operator fusion opens up opportunities for query optimization as well. Most database optimizers treat UDFs as black boxes. Hence, a UDF in a query plan frequently becomes a barrier for query optimization, which could result in missing potentially better execution plans. We argue that fusing UDF and relational operators all together could alleviate to some extent this barrier and provide query optimization with extra opportunities. But this also creates opportunities for operator fusion. A relational operator in between two UDFs may block their fusion. This could be fixed if for example the two UDFs and the relational operator could be fused into a single one. We achieve this functionality by enabling the execution of a relational operator in the UDF execution environment by either implementing the operator in the UDF language or by exporting the internal functions of the database codebase to the UDF environment through a foreign function interface that adds no overhead to the operator execution, but still benefits by allowing fusion to remove all the intermediate execution overheads.

*Pluggable architecture.* We opted for a general solution that can be plugged to existing SQL databases. We provide a dynamic UDF registration mechanism that automatically wraps, embeds, and registers UDFs in a data engine. Expensive overheads stemming from operations such as serialization and deserialization of complex data types (e.g., lists, dictionaries, nested structures) are eliminated at the wrapper layer. Our mechanism supports also stateful execution of the most popular UDF types.

While techniques as operator fusion and JIT compilation have been used to improve SQL query performance [30, 43, 44, 70], they have been studied for queries without UDFs. Approaches that employ fusion via intermediate representations (e.g., LLVM) and compile entire pipelines into single programs that run outside a database, are not suited for in-database execution [15, 18, 79]. Our work is the first to deeply explore how to integrate operator fusion and JIT compilation to optimize UDF queries within SQL databases. Though our pluggable design supports multiple


10.48786/edbt.2026.21

systems, our techniques could be further optimized by integrating directly within a specific engine. In summary, we make the following novel contributions:

(a) A stateful, modular optimization mechanism that is pluggable into popular databases, with most components being engine-agnostic. As a proof of concept, we have integrated QFusor with six database engines.

(b) A holistic operator fusion approach that combines relational operators with all major UDF types (scalar, aggregate, and table UDFs) via a simple yet effective code-blending mechanism.

(c) A cost-based and rule-based optimizer that discovers and creates fusion opportunities in query plans, weighing the trade-offs between UDF overhead and in-database operator execution.

(d) Techniques to address practical but so far overlooked challenges, including handling complex data types (e.g., lists, dictionaries) and reducing serialization/deserialization overhead.

(e) Significant performance gains, achieving up to 40x speedup over state-of-the-art (SOTA) research and commercial systems.

*Outline.* Section 2 discusses related work. Section 3 presents an overview of QFusor. Section 4 describes the UDF registration mechanism and the UDF design specifications. Section 5 presents the QFusor pipeline, the fusion optimizer and code generation. Sections 6 and 7 present our experiments and our final remarks.

## 2 Related Work

*Algebraic style UDF optimization.* An early approach to optimizing queries with foreign functions [11] proposed a declarative rule language to express semantic information about foreign functions and model them as foreign tables. Then query optimization falls back to traditional cost-based techniques (similar to join enumeration) to produce equivalent queries. Follow up work on optimizing queries with user-defined predicates investigated techniques such as pushdown and pullover rules with respect to join trees [12, 13], predicate placement [36], and predicate migration [37]. Although elegant, and conceptually relevant, these early approaches neither suffice to capture the complexity and variety of types of the modern UDFs, nor consider boosting techniques such as JIT compilation.

*UDF integration with the database.* Beside in-engine UDF support in data engines [e.g., 51, 60, 63], research has also dealt with low-level, in-engine optimization of UDF queries. [45] studies strategies to enable dynamic loading of compiled, out-process UDFs in a database, comparing interpreted CPython UDFs against compilation with Cython [5], Nuitka [53], Numba [46], vectorized UDF [44], and multiprocessing parallelization. Our work is closer to efforts considering, as we do, in-process UDF execution that eliminates data import/export between processes [e.g., 27, 65, 72]. [65] integrates NumPy with MonetDB exploiting its vectorized execution. UDO [72] integrates user-defined C++ operators compiled in shared libraries into existing query plans, without considering operator fusion as we do. YeSQL [27] employs a JIT, tuple-at-a-time model for Python UDFs. It supports fusion primarily for scalar UDFs. Our work differs as it presents a principled method to enable fusion of scalar, aggregate and table UDFs along with relational operators, driven by a fusion optimizer. GOLAP [38] considers fusion at a higher-level than our focus, and modifies external pipelines to eliminate overheads derived from (de-)serialization steps.

*Translating UDFs into SQL.* Several approaches aim at translating UDFs to semantically equivalent SQL [16, 19–21, 34, 41, 66, 67, 74]. The related work has focused on various translation aspects, such as (a) rewrite T-SQL scalar UDFs into SQL (loop-less in Froid [66], cursor loops in Aggify [32]), (b) convert PL/SQL with iterations into regular SQL queries by employing a recursive common table expression (PLSQL/AWAY [21]), (c) extend PostgreSQL's JIT compiler to inline lambda expressions in table functions [67], and (d) map algorithmic primitives (e.g., variables, functions, conditions, loops) of procedural languages to PostgreSQL's declarative syntax [7]. Recent systems push the boundaries of UDF to SQL translation: FLummi [24] builds control flow graphs for batch evaluation and parallel execution across DuckDB, Umbra, and PostgreSQL; Franz et al. [29] propose a hybrid strategy to selectively apply inlining or batching based on the UDF's complexity, PRISM [4] introduces UDF outlining and restructuring, enabling optimizations like predicate hoisting and subquery elision, and QURE [73] leverages large language models (LLMs) to translate UDFs to native SQL. The obvious benefit of translating UDF code into SQL is that the translated program enjoys the full power of database optimization and execution. However, the translation is not trivial and it is limited by the scope of the UDF translation (e.g., support for specific packages, libraries). To date, primarily procedural SQL UDFs has been considered. There is also limited support for UDF types and expressions. Our work follows a generic approach targeting the full spectrum of Python programming (i.e., it is not limited to specific packages), hence offering a significantly richer expressivity at the cost of dealing with more complicated query optimization.

*Translating UDFs into an IR.* IR-based approaches rewrite UDFs implemented with specific libraries (e.g., Matlab, NumPy) in an intermediate representation. Supporting only specific libraries poses an important limitation regarding how practical these works can be in real use cases. There are two approaches: employ a custom IR or use generic frameworks (e.g., LLVM) as an IR.

The fist category focuses on issues such as: (a) data movement optimizations for data-parallel operators, e.g., relational, linear algebra (Weld [57]), (b) array-based applications with MATLAB UDFs (HorsePower [14, 35]), (c) extract SQL with two IRs, one for applications and one for functional representation, and check for simple optimizations as push down computations [22], (d) investigate cost-based optimization for fusing plans over DAGs of linear algebra operations to produce Java code for each operator [8], (e) analyze Spark plans, perform relational processing with frameworks like Tensorflow, and execute on custom runtime (Flare [23]), (f) deal with polyglot queries (Java, Javascript, Python) and perform IR-based fusion of built-in operators and UDFs (Babelfish [31]), (g) translate Pandas to IR and then to C++ (SDQL.py [68, 69]), and (h) generate a statically-typed IR from high-level semantics of Python UDFs [42].

The second category focuses on compiling end-to-end Python pipelines into a single program, e.g., using LLVM compilation [47]. Two representative systems in this area are Tuplex and Tupleware. Tuplex [79] is a data analytics framework utilizing a performant end-to-end, tuple-at-a-time, prototype tracing JIT compiler. A developer creates a pipeline using a series of high-level, LINQ-style operators[49] (e.g., map, filter, join) and passes UDFs as parameters to these operators. It supports operator fusion via LLVM IR. In Tupleware [17, 18], a developer defines workflows in a host language (e.g., C++, Python) by passing UDFs to API operators similar to Apache Spark[78] or DryadLINQ[88]. Tupleware compiles the entire workflow, including UDFs, into a fully compiled, self-contained program, which is then deployed for execution. In doing so, it blends high-level optimization (e.g., operator pushdown, join reordering) with low-level compiler techniques (e.g., inline expansion, SIMD vectorization) using LLVM to provide

a language-agnostic front-end for map-reduce style operators and introspect UDFs [9, 39, 40]. But it only supports UDFs for which types can be inferred statically, only numerical types, no polymorphic types (e.g., NULL) [79]. Note also that both systems lack support for table returning UDFs.

A critical difference from IR-based efforts is that we focus on the fusion of relational and procedural UDF operators directly without any intermediate medium. And our generated UDF query runs in a SQL database. Low-level compilation techniques are also considered in other works, e.g., Tuplex, Tupleware. But our architecture is very different, as we deal with the non-trivial impedance mismatch between two different execution environments: Python and the database (usually, C/C++). For example, Tuplex does not have to deal with data/type conversions or serialization overheads, as the entire pipeline runs in the same context, making it an excellent baseline as it offers presumably the best possible execution of (out-of database) Python pipelines. Tupleware and Tuplex are end-to-end systems that compile the entire pipeline into a single program and then execute it, relying on their optimization and execution capabilities. Our work follows an engine agnostic design to integrate a UDF execution environment into a SQL database. It compiles each UDF separately and then integrates it to the query plan. This non-intrusive way to optimize UDF execution leaves query optimization and execution to the SQL database. It is also worth noting that even if one tried to integrate the logic of Tuplex and Tupleware into a database, not all engines support LLVM and LLVM introduces substantial compilation latencies, which are especially crucial for short queries [30, 52]. We investigate this aspect further in the experimental evaluation in Section 6.4.5.

To the best of our knowledge, QFusor is the first in-database UDF integration approach that (a) considers JIT-compiled fusion of all popular UDF types scalar, aggregate, table (prior art has focused primarily on scalar functions) along with relational operators, (b) offers a principled, optimization strategy to operator fusion, and (c) can be plugged into existing SQL database engines.

# 3 The Case for QFusor

## 3.1 Example UDF Query

Consider a real-world analytics use-case on publication and funding data, aiming to assess how research projects influence collaboration among scientists. The data, collected from sources such as PubMed, arXiv, crossref, and Zenodo, needs cleansing and homogenization. The query (see Figure 1) processes each publication to extract author pairs, then computes per project the number of author pairs who collaborated during the project, before it began, and after it ended.

The query contains several cleansing UDFs to (a) avoid false negatives, (b) normalize authors names (`lower`) and remove short terms, (c) sort the authors, (d) flatten author pairs to multiple rows (`combinations`), (e) retrieve funding information, and (f) standardize dates (`cleandate`). Finally, it retrieves all publications authored by each author pair and aggregates the result by funder, funding class, and project. This query suffers from several UDF related overheads. Authors are stored in JSON, which requires multiple (de-)serializations to process it. The query is UDF-heavy, with expensive operators such as regular expressions (`removeshortterms`), string sorting (`jsort`, `jsortvalues`), and author pair generation with a table UDF (`combinations`). Frequent C↔Python context switches between the database and UDF execution environments add further overhead, especially, in the

```
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;
```

**Figure 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
));
```

**Figure 2: Example fused UDF query**

lower part of the query that involves scalar operations with filters, case expressions, and aggregations.

## 3.2 Overview of QFusor

Figure 3 illustrates an overview of the QFusor architecture. QFusor connects as a plugin to a SQL database, either embedded (e.g., SQLite, DuckDB) or server-based (e.g., MonetDB, PostgreSQL). It requires that the database supports: (a) a plan generation mechanism (e.g., a query optimizer), and (b) a UDF registration mechanism and support for C UDFs. UDF developers create their user-defined functions and register them in the database. At query time, the user submits a SQL query. QFusor adds a client layer to interact with the database (shown in the figure as a thin blue line left and right of the DBMS). If the query involves UDFs, the client propagates the plan produced by the optimizer to QFusor. Queries without UDFs are processed as usual. QFusor comprises a pipeline of four steps.

*Discover fusible operators.* First, it analyzes the query plan to discover UDFs and relational operators that have data dependency to each other and could potentially be fused together (see Section 5.1).

*Fusion optimization.* Then, it employs rule-based and cost-based techniques to decide what operators should be fused (see Section 5.2).

*JIT code generation.* Next, it generates just-in-time the code implementing the new execution plan and registers the produced fused UDF(s) as new UDF(s) in the UDF registry (see Section 5.3).

*Query rewrite.* Finally, it rewrites the SQL query by replacing the affected operators in the query with the newly produced fused UDF(s). It produces either a rewritten SQL statement that is resubmitted to the database (path 1) or, for selected data engines, a valid execution plan that is sent directly to the execution engine (path 2). In the former case, in our current implementation, the query does not undergo another QFusor process. This scenario would be helpful in case the optimizer reorders the operators in the plan enabling thus additional fusion opportunities, an interesting direction for future work (see Section 5.4).

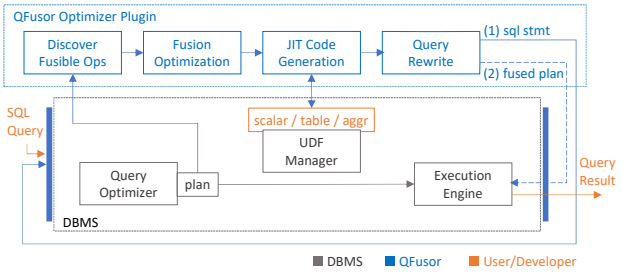



**Figure 3: Pluggable architecture of QFusor**

*Example Revisited.* The example query presents various fusion opportunities. Several of its UDFs could be fused, such as `extractid`, `extractfunder`, `extractclass`, `lower`, `jsortvalues`, `removeshortterms`, `jsort`, and `combinations`. Another option is to fuse the functions `cleandate` and `sum` with the relational operators, i.e, `filters` and `case`, which can be JIT implemented as UDFs and run in a fused UDF. Figure 2 shows a fused version of the query (see also Section 6.4.1).

# 4 Enabling QFusor

There are two key enablers for QFusor: the UDF registration mechanism and the UDF design specifications.

## 4.1 UDF Registration

The QFusor client employs a UDF registration mechanism, which is responsible for automatically wrapping, embedding, and registering the UDFs to the data engine. When the UDF developer submits a new UDF decorated with its type (i.e., scalar, aggregate, table) and its output data types, a function that wraps the UDF is automatically created by our registration mechanism. The wrapper function converts the input data to C data objects, includes a call to the UDF submitted by the developer, and finally assigns the results to a C data object, which is accessible to the data engine. The wrapper function is then compiled into dynamically loadable objects that are embedded into a C UDF. The latter is registered in the data engine via a `CREATE FUNCTION` statement which is also automatically generated by our registration mechanism.

As an example, let us examine the `lower` function of the running example. The UDF developer submits the following UDF:

```
@scalarudf
def lower(val)->str:
    return val.lower()
```

The UDF registration mechanism wraps the UDF as follows:

```
def lower_wrap(input,size,result):
  for i in range(size):
   inval = ffi.string(input[i])   <- convert input c data object
    resval = lower(inval)          <- call original UDF
    result[i] = ffi.from_buffer(resval) <- convert result
```

Here, `ffi` is a foreign function interface that is used to convert the data format between the data engine and the UDF's execution environment. Next, `lower_wrap` is compiled into a shared library and is registered in the data engine via a `CREATE FUNCTION` statement, as a C UDF following the data engine's specifications.

In this example, the registration mechanism wraps and registers a single UDF. The same mechanism is used to generate functions that loop-fuse multiple pipelines of UDFs (see Section 5.3.1). The mechanism that creates this wrapper function is called when a new UDF or a fused UDF is registered and requires the following parameters: (a) the UDF object or a list of UDF objects ordered by their data dependencies (discussed shortly), (b) the input arguments and their data types, (c) the return data types, (d) the UDF types (i.e., scalar, aggregate, table). Then, it generates the wrapping code according to the UDF design specifications presented next.

## 4.2 UDF Design Specifications

To enable effective fusion strategies, we expect the UDFs to abide by the design specifications we describe next.

*4.2.1 Scalar UDFs.* A scalar UDF returns a single value for each row of data it reads. It can get one or more arguments (treated as a row of data), and returns one value as the result of the Python function onto the processed row. An example scalar UDF follows.

```
def inc1(x):
  return x + 1
```

*4.2.2 Aggregate UDFs.* An aggregate function is invoked once per group (i.e., a set of rows) produced by an external grouping operator, it performs an operation on this set of rows and returns one value. We adopt the incremental init-step-final model, which avoids group materialization. This model works by initializing a state (init), then updating the state as each successive input row is processed (step) until the whole input is consumed, and finally, returning a single value (final). We implement aggregate UDFs as Python classes. An example summation aggregate can be formed as follows.

```
class SumAggregate:
  def __init__(self):    <- initialize state
    self.s = 0
  def step(self, v):     <- update state
    self.s += v
  def final(self):       <- return the final state
    return self.s
```

*4.2.3 Table UDFs.* A table UDF reads one or more arguments (treated as a row of data), and returns zero or more rows consisting of one or more columns. To enable fusion and loop fusion [28, 70] for table UDFs, we implement fully pipelined table UDFs using Python generators, which produce a lazy iterator that does not materialize data and use yield instead of return to preserve the function state. The generator is suspended and resumed after yield, which boosts pipelines for large datasets. Consider the example:

```
def tuple_gen(inp_datagen, *args):
  for inp in inp_datagen(*args):
    yield outp[0], outp[1]
```

`tuple_gen` is a generator function modeling a table UDF that takes an arbitrary number of input columns (`*args`), performs an operation, and outputs a table with two columns (`outp[0]`, `outp[1]`). `inp_datagen` is also a generator function that provides input rows to the table UDF. For example, assuming that this table UDF inputs two columns, `inp_datagen` would be as follows.

```
def inp_datagen(col1, col2, count):
  for i in range(count):
    yield col1[i], col2[i]
```

This approach allows the developer to iterate over the input columns once, without requiring materialization. However, if a table UDF requires a fully materialized column, then we convert the generator into a materialized list. An example follows.

```
mat_input = list(inp_datagen(*args))
```

These are the most frequently supported UDF types in modern data engines [e.g., 61, 81, 85]. Supporting additional UDF

types, such as user-defined analytical functions or window functions that allow advancing the input and the output processing independently, is an interesting future direction.

*4.2.4 Complex data types.* UDFs typically support a limited set of datatypes aligned with the underlying data engine. However, Python pipelines often require complex data types like lists or dictionaries. Most databases represent complex datatypes as JSON objects, which introduce a (de-)serialization overhead. Our approach allows UDFs to return complex Python data types (e.g., lists, dictionaries) without serialization. The serialization is handled by the wrapping code that interacts with the database. This presents an optimization opportunity: the (de-)serialization steps can be eliminated at the wrapper layer, leading to a substantial reduction in execution time.

Let us consider an example. The snippet on the left shows a typical Python UDF implementation. The snippet on the right shows an implementation following our specifications. Here, we assume annotated types, but we also support dynamic types and type definition at query time (details are omitted for space considerations).

```
def tokens(val) -> str:
  return json.dumps(val.split())
def counttokens(val) -> int:
  return len(json.loads(val))
```

```
def tokens(val) -> list:
    return val.split()
def counttokens(val) -> int:
    return len(val)
```

The wrapping code produced for `tokens` would be:

```
@ffi.def_extern()
def wrapper_tokens(input, insize, result):
   ts = [tokens(ffi.string(input[i])) for i in range(insize)]
   for i in range(insize):
      result[i] = ffi.from_buffer(memoryview(serialize(ts[i])))
```

In a fusion case (e.g., `counttokens(tokens(val))`), the wrapper function is JIT created removing (a) (de-)serialization calls, (b) excessive C to JIT conversions, and (c) materialization of intermediate results.

```
@ffi.def_extern()
def wrapper_counttokens_tokens(input, insize, result):
   for i in range(insize):
      result[i] = counttokens(tokens(ffi.string(input[i])))
```

*4.2.5 DML queries.* SQL databases routinely support UDFs in DML queries (inserts, updates, deletes), as for example:

```
update table set col1=udf1(col2) where udf2(udf3..udfN(col1,..,colN))
```

Our techniques apply to DML queries with UDFs as well, similar to our handling of UDF queries. This critical functionality is not supported by SOTA approaches (e.g., PySpark, Tuplex).

## 5 The QFusor Pipeline

The QFusor pipeline operates at runtime on queries containing UDFs. For such a query, our client probes the optimizer using an `EXPLAIN` statement and propagates the plan generated to the QFusor pipeline (see also Sect. 3.2). Next, we present this pipeline.

### 5.1 Discover Fusible Operators

*5.1.1 Fusibility conditions.* Two operators in a query plan have a *data dependency*, when the output of the first is propagated to the input of the second. Formally, the dependency between two consecutive operators $o_1$ and $o_2$ is defined by the Bernstein condition [6]:

$$[o_1.\texttt{in} \cap o_2.\texttt{out}] \cup [o_1.\texttt{out} \cap o_2.\texttt{in}] \cup [o_1.\texttt{out} \cap o_2.\texttt{out}] \neq \varnothing$$

And in particular, if $[o_1.\texttt{out} \cap o_2.\texttt{in}] \neq \varnothing$ holds, then there is a data dependency (a.k.a. read-after-write, RAW) between $o_1$ and $o_2$, denoted as $o_1 \rightarrow o_2$. Operators that have a data dependency are candidates for fusion, and we refer to them as *fusible operators*. If two fusible operators, $o_1$ and $o_2$, are fused into a single operator $o_f$, we write $o_1 \rightarrow o_2 \Rightarrow o_f$, where the input and output of $o_f$ are $o_1.\texttt{in}$ and $o_2.\texttt{out}$, respectively. We refer to $o_f$ as a *fused operator*. Two or more consecutive fusible operators in a plan form a *fusible section*. A plan may contain more than one fusible section.

**Algorithm 1** Create DFG

**Require:** Query plan $Q = (V_Q, E_Q)$
1: $G \leftarrow \{\}$
2: **for all** $v \in V_Q$ **do**
3:  **for all** $u \in V_Q$ **do**
4:  **if** $v = u$ **then** continue **end if**
5:  **if** Bernstein condition holds for $u$, $v$ **then** $G \leftarrow (u, v)$ **end if**
6:  **end for**
7: **end for**
8: **return** DFG $G$

In our approach, we treat both UDFs and relational operators as potentially fusible operators. And under some conditions, we enable fusion between both UDFs and relational operators (see Section 5.3). Currently, we support the following cases of operator fusion involving UDF (`udf`) and relational (`rel`) operators:

(F1) `udf`→`udf` ⇒ `f_udf`
(F2) `udf`→`rel` ⇒ `f_udf` (or `rel`→`udf` ⇒ `f_udf`)
(F3) `udf`→`rel`→`udf` ↬ `rel`→`udf`→`udf` ⇒ `rel`→`f_udf`

The first case involves two UDFs with a data dependency, which we fuse into a single UDF `f_udf` that combines their functionality. The type of the resulting UDF depends on the types of the original functions. The cases of UDF fusion we support, with the types of UDFs composed and returned, are listed in Table 2.

The second case involves encapsulating a relational operator into a Python UDF, allowing it to be fused with UDFs it has a data dependency with. This can remove optimization barriers caused by black-box UDFs in the query plan. A promising future direction is to explore the reverse transformation: express a UDF's functionality using one or more relational operators [7].

The third case introduces a commutative property in our operator fusion approach. By analyzing UDF code alongside operator schemata and types (e.g., scalar, aggregate), we determine when operator reordering (↬) is valid. Building on prior formal definitions [75–77], we conservatively allow reordering only when operators do not operate on the same fields. Consider a schema `(a,b,c)`, two scalar UDFs $u_1$`(a,b)` and $u_2$`(a,b,c)` operating on the fields `a,b` and `a,b,c`, respectively, and a filter `flt(c)` on the field `c`. The query plan $u_1$`(a,b)`→`flt(c)`→ $u_2$`(a,b,c)` is semantically equivalent to `flt(c)`→ $u_1$`(a,b)`→ $u_2$`(a,b,c)`. Future work could apply techniques such as symbolic execution to unravel more reordering opportunities [40, 84, 87]. While operator reordering is a query optimizer's function, our goal with F3 is twofold. (a) Enable additional fusion opportunities, as reordering can eliminate blockers between fusible UDFs. (b) Assist query optimization by enabling optimizations such as filter push-down [87] or UDF push-up [37], which are otherwise impossible if UDFs are treated as black boxes.

*5.1.2 Discover fusible operators.* Drawing from compiler theory, we model the problem of identifying fusible operators in a query plan as a *dependence analysis* problem and construct a data flow graph (DFG) to capture operator dependencies [2]. The process is formally described in Algorithm 1.

## 5.2 Fusion Optimization

Fusion Optimization (FO) uses a hybrid cost-based and rule-based strategy to decide which operators to fuse. Its primary goal is to reduce the overhead from the interaction between SQL's relational processing and the procedural execution of UDFs.

*5.2.1 Discover fusible sections.* To achieve this, we use dynamic programming to traverse the DFG, identifying patterns that match fusion cases (F1)–(F3). This approach can be extended to support additional fusion patterns. The result of this step is a set of fusible sections. Note that in some queries the DFG may be the entire plan.

The process is formally described in Algorithm 2. First, we perform a topological sort of the DFG to respect dependencies. Then, we check whether an operator should be considered for fusion. Note that operators that are too expensive or even infeasible to fuse carry an infinite cost. The core of the algorithm (ln: 8-16) attempts to extend previously computed fusible sections by appending more operators following the fusion cases (F1)-(F3). For an operator $v$, we examine its producers $u$. If $v$, $u$ are either fusible or could be reordered (`FusibleOrReorderable`), we explore whether $v$ could be added to the section containing $u$ and form a new section $S$. If $S$ is valid, i.e., contains consecutive fusible operators, (`IsValidSection`) and its cost could have a potential gain compared to not fusing $v$, we identify the most promising permutation for $S$ according to reordering opportunities guided by the cases (F1)-(F3) (`OptimPermutation`). The decisions are driven by a function $F$ that evaluates the cost of fusing a sequence of operators (discussed shortly). Finally, we produce maximal non-overlapping sections ready to be fused. The algorithm is correct as data dependencies are preserved due to the use of Bernstein conditions and reordering constraints. Its worst-case time complexity is exponential in the length of the fusible sections, due to checking all valid permutations (factorial in section length). But in practice, memoization, bounded dynamic programming steps, and heuristic pruning make the algorithm efficient.

*5.2.2 Operator cost analysis, $F(v)$.* The cost of relational operators is computed using engine-provided statistics, such as row estimates and cost units derived from the query plan.

The UDF cost encapsulates two quantities. The *processing cost* of the UDF itself and the *wrapping cost*, the cost of the wrapper function that handles data copying, transformation, and materialization to interface with the data engine. Since the wrapper functionality is concrete and measurable, its cost can be accurately estimated. A key objective of FO is to minimize at least this wrapping overhead.

Estimating the UDF processing cost is more challenging, as most data engines provide little or no statistics. Some engines allow developers to supply cost-related metadata during UDF registration (e.g., via `CREATE FUNCTION` options for estimated cost and row count [62]. In many cases though, the UDF cost is unknown. To address this, FO maintains a lightweight dictionary of average execution statistics for UDFs, such as execution time and selectivity. For scalar and aggregate UDFs, selectivity is known: the output size of a scalar UDF equals its input size, and aggregate UDFs return a single value. FO gradually builds a cost model by learning from past UDF executions, using coarse-grained estimate buckets instead of precise values. It profiles UDFs using Bayesian Optimization, a method well-suited for tuning black-box functions with limited test runs, inspired by CherryPick [1]. This adaptive process is facilitated by the stateful implementation of the UDF mechanism, which helps keep track of the collected statistics at UDF runtime. For new UDFs lacking data, FO falls back on heuristics (discussed next) to avoid cold starts. Over time, as more data is collected, the cost model becomes more accurate, balancing exploration and exploitation, as typically happens in learning tasks. Note that the choice of a particular cost model is orthogonal to our approach, as different models may be preferable for different applications or engines. Accordingly, QFusor could integrate any alternative cost modeling approach (e.g., GNN based UDF cost estimation [86]).

**Algorithm 2** Discover Fusible Sections

**Require:** DFG $G = (V, E)$, cost function $F$
1: $V_{\text{sorted}} \leftarrow \text{TopologicalSort}(G)$
2: $dp[v] \leftarrow \infty$, $section[v] \leftarrow [\,]$ for all $v \in V$ ▷ Initialization
3: **for all** $v \in V_{\text{sorted}}$ **do** ▷ Update
4: **if** $F(\{v\}) < dp[v]$ **then**
5: $dp[v] \leftarrow F(\{v\})$ ▷ Min cost of a section ending at $v$
6: $section[v] \leftarrow [v]$ ▷ Operators forming this section
7: **end if**
8: **for all** $u \in \text{Predecessors}(v)$ **do**
9: **if** $\text{FusibleOrReorderable}(u, v)$ **then** ▷ Check fusion cases
10: $S \leftarrow section[u] + [v]$
11: **if** $\text{IsValidSection}(S)$ and $F(S) < dp[v]$ **then**
12: $dp[v] \leftarrow F(S)$ ▷ (Compute potential gain)
13: $section[v] \leftarrow OptimPermutation(S)$ ▷ Check (F3)
14: **end if**
15: **end if**
16: **end for**
17: **end for**
18: $Visited \leftarrow \emptyset$, $Sections \leftarrow [\,]$
19: **for all** $v \in \text{reverse}(V_{\text{sorted}})$ **do** ▷ Section selection
20: **if** $section[v] \cap Visited = \emptyset$ **then**
21: $Sections \leftarrow Sections \cup section[v]$
22: $Visited \leftarrow Visited \cup section[v]$
23: **end if**
24: **end for**
25: **return** $Sections$ ▷ Ready for code-gen

*5.2.3 Section cost analysis, $F(S)$.* Starting from the fusible sections identified in the discovery step, FO determines which operators within each section should be fused. There are two main cases (F1) and (F2); with (F3) reducing to (F1) after reordering. In (F1), which involves only procedural UDFs, FO always recommends fusion as (a) it eliminates at least the wrapping cost, and (b) it produces longer traces for the tracing JIT compiler, enabling better runtime optimization. In (F2), which involves both UDFs and relational operations, fusion decisions are more complex.

FO considers two execution strategies for relational operators. Executing them in the UDF environment to avoid intermediate result materialization and enable optimizations such as loop fusion. Or executing them in the data engine, which may offer a more optimized implementation. The decision relies on two aspects: (a) Operator complexity, performance-critical or complex operators are better handled by the engine. (b) Selectivity, operators with low selectivity (returning many values) could benefit from in-UDF execution, as the more data the operator processes, the larger the fusion benefit is due to the eliminations of data copy and transformation.

Hence, in an (F2) case FO considers the following factors: (a) the cost of a relational operator if executed in the data engine, (b) its cost if it runs in the UDF execution environment, (c) the overall benefits for the fusible section (e.g., a relational operator

| | |
|---|---|
| $\|u\|, \|u_f\|, \|r\|$ | input cardinality for UDFs, the fused UDF, and $r$ |
| $\sigma_u, \sigma_{u_f}, \sigma_r$ | selectivity estimates for UDFs, the fused UDF, and $r$ |
| $W_{in}, W_{out}$ | cost per tuple for the wrapper functions' input and output |
| $C_r, C_{ru}$ | cost per tuple for $r$ if executed in the data engine or the UDF environment |

**Table 1: Notation used in the cost model**

in between two UDFs may block their fusion, but if it runs on the UDF environment, presumably all three operators can be fused into one). Collectively, these factors formulate the following inequality that FO uses to determine how to treat the fusion of UDF and relational operators: if the inequality holds for a given relational operator $r$, then we execute $r$ in the UDF environment (see also Table 1).

$$\sum_{u=1}^{N}(|u|*(W_{in}+W_{out}*\sigma_u)) - |u_f|*(W_{in}+W_{out}*\sigma_{u_f}) > |r|*(C_{ru}*\sigma_r - C_r*\sigma_r)$$

The inequality checks if the gain of running $N$ UDFs in isolation vs. as a single fused UDF (left) is greater than the loss of running $r$ in the UDF environment vs. in the data engine (right). If the right portion of the inequality is negative (i.e., it is a gain, not a loss) then we run $r$ in the UDF environment. The number $N$ of UDFs considered here is the maximum set of UDF operators affected by $r$ in the fusible section we examine.

*5.2.4 Heuristics.* To mitigate cold-start issues with newly registered UDFs, FO applies a set of heuristics; some generic, others engine-specific (not shown here). These are derived from common practices and extensive experimentation. They are especially valuable for rule-based engines that lack cost-based optimization. Due to space constraints, we list here a few example heuristics:

- Fuse all fusible scalar, aggregate, and table UDFs.
- If there is a data dependency between a relational *filter* and a UDF (or multiple UDFs), fuse the filter with the UDF(s) if the filter is not highly selective; e.g., it filters out less than 20% of its input.
- Fuse *group-by* operators if possible. (We elaborate on aggregate UDFs and group-by later.)
- Fuse a *distinct* operator, if it is highly selective (e.g., filters out more than 90% of its input), otherwise do not.
- Avoid fusing *join* and *sort* operators; the performance gain is typically minimal.

## 5.3 JIT Code Generation

Next, we generate the code to implement the fused UDFs and relational operators, considering standard optimizations such as *function inlining* and *loop fusion* to avoid intermediate result materialization and redundant looping across all UDF types. The fused logic is JIT-implemented inside a wrapper function, which is then JIT-compiled and registered as a new UDF in the database (Section 4.1).

Empowered by the UDF design specifications, the JIT creation of loop fused UDFs is a deterministic process. UDF fusion takes place inside the main `for-loop` of the wrapper function. This loop coordinates the UDFs according to the query plan.

Loop fusion is supported across all types of operator fusion. For aggregate UDFs, the init-step-final model enables pipelined execution, while table UDFs use a generator-style input/output model to achieve the same (see Section 4.2). However, if a UDF includes a blocking operation (e.g., table transpose, median), materializing input becomes necessary, and loop fusion no longer applies. Table 2 illustrates the applicability of loop fusion across all fusion scenarios. One special case is Expand, a variant of table UDFs that consumes a single tuple at a time and returns multiple rows per input tuple [87]. It is always loop-fusible and typically follows a scalar or aggregate operator to split its result into multiple rows.

*5.3.1 Loop Fusion Implementation.* Next, we present how fusion is JIT-implemented for various combinations of UDF types, using references from our running example.

*Scalar - Scalar.* Let us consider the following example SQL query:

```
SELECT removeshortterms(lower(authors)) FROM pubs;
```

`lower` and `removeshortterms` are two scalar UDFs that operate on the same field `authors`. Since they are data dependent UDFs they can be executed inside a single `for-loop`. In the wrapper function, the part of the loop which fuses the two UDFs using here the (TF1) template would be as follows:

```
for i in range(tuples):
    resultval = removeshortterms(lower(authors[i]))
```

When this code is compiled with the tracing JIT compiler, the compiler automatically inlines and vectorizes function calls before locating hot traces for JIT compilation. Hence, this code snippet does not involve function call overheads. Additional scalar UDFs can be fused if they are added in the pipeline inside the `for-loop`.

*Scalar - Aggregate.* Let us consider the following example SQL query:

```
SELECT funder, countauthors(removeshortterms(authors))
FROM pubs GROUP BY funder;
```

`countauthors` is an aggregate UDF, and `removeshortterms` is a scalar UDF. Since both UDFs operate on the same column they can be fused into a single loop. Using the (TF2) template, the fusion code in the wrapper function would be as follows:

```
aggrs = [countauthors() for _ in range(g)]
for i in range(tuples):
    aggrs[aggr_group_data[i]].step(removeshortterms(col[i]))
for i in range(g):
    result[i] = aggrs[i].final()
```

`g` is the number of groups and `aggr_group_data` is a C-array of type `size_t` indicating the group assignment for each row. We instantiate `g` aggregate UDFs, one per group, and apply any number of scalar UDFs to each row of the table inside the `step` method, before updating the state of aggregate UDFs. Finally, we populate the result array with the final output from each aggregate.

*Scalar - Table.* Loop fusion is possible in this case, as shown in (TF3) and (TF5) in Table 2, because table UDFs are implemented as Python generators with lazy iterators to avoid a full materialization of the output column. Using the (TF5) template the fusion of scalars executed on the result of a table UDF takes place during the materialization of the result array. On the other hand, for scalar UDFs executed before a table UDF (TF3), loop fusion of scalar UDFs occurs within the lazily evaluated generator function that implements the table UDF; see also Section 4.2.

*Aggregate - Table.* In this case, loop fusion is possible if there is no group-by between the UDFs. The fused UDF will resemble the UDF in Scalar-Table case. If there is a group-by, then loop fusion is not applicable as the output column of the table UDF is processed independently of the aggregate UDF in order to calculate the groups.

| | (TF1) | (TF2) | (TF3) | (TF4) | (TF5) | (TF6) | (TF7) | (TF8) |
|---|---|---|---|---|---|---|---|---|
| **1st operator** | scalar | scalar | scalar | table | table | table | aggregate | aggregate |
| **2nd operator** | scalar | aggregate | table | table | scalar | aggregate | scalar | table |
| **result** | scalar | aggregate | table (or scalar+expand) | table | table | aggregate | aggregate | table (or aggregate+expand) |
| **fusion** | for tuple in input:<br>  return sc2(sc1(tuple))) | aggr.init()<br>for tuple in input:<br>  aggr.step(sc(tuple))<br>return aggr.final() | def inputgen(sc, input, count):<br>  for i in range(count):<br>  yield sc(input[i]))<br><br>datagen = inputgen(sc, input, count)<br>for tup in tbl(datagen):<br>  result.append(tup) | def inputgen(input, count):<br>  for i in range(count):<br>  yield input[i]<br><br>inputgen = inputgen(input, count)<br>tblgen = tbl2(tbl1(inputgen))<br>for tuple in tbl2gen:<br>  return tuple | def inputgen(input, count):<br>  for i in range(count):<br>  yield input[i]<br><br>inputgen = inputgen(input, count)<br>for tuple in tbl(inputgen):<br>  return sc(tuple) | def inputgen(input, count):<br>  for i in range(count):<br>  yield input[i]<br><br>aggr.init()<br>for tuple in tbl(inputgen(input,count)):<br>  aggr.step(tuple)<br>return aggr.final() | aggr.init()<br>for tuple in input:<br>  aggr.step(tuple)<br>return sc(aggr.final()) | aggr.init()<br>for tuple in input:<br>  aggr.step(tuple)<br>tblgen = tbl(aggr.final())<br>for tuple in tblgen:<br>  return tuple |
| **loop fusion** | yes | yes, unless aggregate materializes its input | yes, unless tbl materializes its input | yes, unless tbl2 materializes its input | yes | yes, unless aggregate materializes its input | yes | yes, unless tbl materializes its input |

Legend: sc{1,2}: scalar function object | aggr: object of aggregate class | tbl{1,2}: table function object | tblgen: table returned generator | inputgen: input data generator

**Table 2: Loop fusion templates**

*5.3.2 Fusion of relational operators.* Relational operators, like UDFs, can be fused based on their semantics. We classify each operator as scalar, aggregate, or table-returning, depending on its input and output. This classification guides how they are fused with various UDF types. After rewriting the operator code in Python, we apply the fusion rules from Table 2 and inline the code into the fusion wrapper function. Table 3 lists example relational operators supported in QFusor, along with their types, their input/output, and whether they are loop fusible with other operators. Outputs include bool, single row, or resultset (full table or nested subquery results).

The operators listed in Table 3 are *fusible*, i.e., they may be fused with other operators. Whether they are also *loop fusible* would enable a performance optimization allowing the fused operators to be executed in the same hot-loop [70] (see also Section 6.4.3). In general, pipelined operators are loop fusible (e.g., filter, join), whereas blocking operators (e.g., group, order) are not. However, if a blocking operator is last in a series of operators that can be fused all together, then it may participate in the same hot-loop, as long as it can consume its input in a pipelined fashion (e.g., order vs. pivot).

Fusion of relational operators is achieved through the UDF registration mechanism (see Section 4.1). We support two approaches: (a) rewriting (a.k.a. offloading) the relational operator in the UDF's language, and (b) via a call to the database source code, for engines offering this option [10]. Next, we explore representative cases of both mechanisms: `filter` and `distinct` as in-UDF implementation, and `group-by` as a case of an external call to the data engine.

*Filter.* A `filter` can be modeled as a scalar UDF that returns a single boolean value for each row. It evaluates to true if the row meets the filter condition, and false otherwise. Consider the following example query:

```
SELECT udf2(udf1_col)
FROM  ( SELECT udf1(col1) as udf1_col FROM table )
WHERE udf1_col != 10;
```

In this case, `udf1` and `udf2` are fusible only if the filter between them is also fused. Fusion of the relational operator is done as follows:

```
def not_equal(val, noteq):
    True if val != noteq else False

for i in range(tuples):
    udf1_res = udf1(col[i])
    if not_equal(udf1_res, 10):
        result[j] = udf2(udf1_res)
        j += 1
```

*Distinct.* `Distinct` can be modeled as a table UDF, which inputs a number of columns and outputs a number of columns, often with differing sizes. It can be implemented in the UDF's language and JIT-compiled to enable fusion with other UDFs. Consider an example:

| relational operator | type | loop fusible | input / output |
|---|---|---|---|
| filter | scalar | yes | row → bool |
| inner join | scalar | yes | row1, row2 → bool |
| distinct | table | yes | resultset1 → resultset2 |
| case | scalar | yes | row → row |
| order by | table | no | resultset1 → resultset2 |
| group by | table | no | resultset1 → resultset2 |
| pipelined aggregates (e.g., count, sum) | aggregate | yes | resultset → row |
| blocking aggregates (e.g., median) | aggregate | no | resultset → row |
| union all | table | yes | resultset1, resultset2 → resultset |
| union/intersect/except | table | no | resultset1, resultset2 → resultset |
| arithmetic (e.g., +-%*) | scalar | yes | row → row |
| pivot/unpivot | table | no | resultset1 → resultset2 |
| is null / is not null | scalar | yes | row → bool |

**Table 3: Loop fusion for relational operators**

```
SELECT distinct col1 FROM SELECT udf(col1) AS col1 FROM table;
```

In this case, the distinct operator is fused with the udf as follows:

```
distinctset = set()
for i in range(tuples):
    distinctset.add(udf(col[i]))
```

*Group-by.* `Group-by` can be modeled as a table UDF returning two columns: one with the grouped attribute and another with integer pointers indicating each tuple's group assignment. `Group-by` fusion is implemented via a call to the database source code through a foreign function interface (FFI), enabling its use within the tracing JIT. Since this approach is database-specific, we describe it here using MonetDB. MonetDB provides a C API for UDFs, supporting dynamic creation, compilation, linking, and execution of C-UDFs, as well as in-process UDF execution. We extended this API to export internal MonetDB functions that handle grouping, BATs (MonetDB's data columns) initialization, value insertion, and memory storage/loading of BATs. These functions are invoked from within the tracing JIT to allow direct interaction with MonetDB at runtime. We showcase this process using an example SQL query that involves a table UDF (`tudf`) and an aggregate UDF (`audf`):

```
SELECT audf(t.tcol)
FROM tudf( (SELECT col FROM table) ) AS t
GROUP BY t.gattr;
```

To fuse the aggregate UDF `audf` with the table UDF `tudf`, we must execute a group-by on `gattr` within the fused UDF. Implementing group-by in Python would be an option, but this would be an inefficient operation that negates the performance benefits of tracing JIT, UDF fusion, and loop fusion. Using Python C foreign function interface (CFFI), we enable Python code to interact with external C code, and here, with MonetDB's shared libraries. In QFusor, we leverage this by exporting MonetDB's internal functions (e.g., to create, modify, and group BATs), hence enabling the execution of group-by operations from a Python UDF with no additional overhead, whilst we still leverage database optimizations (e.g., indexes, cached data). Since we operate

on raw memory pointers and access MonetDB's internal stack through exported functions, neither the data nor the functions on it ever leave the engine. The same strategy applies for all relational operations (e.g. join, order-by). Naturally, the optimizer may decide not to fuse some relational operators. For example, join will likely run in the data engine in most cases.

Note that this functionality is applicable to engines supporting in-process execution of C UDFs, either embedded or server DBMSs such as MonetDB and Vertica (with fenced mode). In QFusor, wrapper functions run in the same process as the data engine, eliminating inter-process communication overheads between the engine and UDFs. To ensure robustness, these wrapper functions invoke the UDF logic within a try-except block, allowing graceful handling of potential UDF execution errors.

*Preserving semantics.* Modeling a relational operator as a procedural UDF requires careful handling to ensure semantic equivalence. Currently, we rewrite simple operators as UDFs, where maintaining correct semantics is safe. When we use a call to the operator's data engine source code, the operator runs in the engine's context, which has no implication with the semantics. Clearly, for production-level maturity we need to fine tune issues such as rounding, type conversions, casts, operator precedence, null-handling, and so on.

## 5.4 Query rewrite

The QFusor pipeline completes with creating a new query comprising the fused operator(s) that replace the operators that have been fused. All necessary UDFs, including the newly fused ones, have already been created, compiled, and registered to the SQL database. The task at hand is to perform query rewrite. QFusor offers two options. The first is to create a new SQL statement and submit it to the database. An alternative database specific option is to generate a new execution plan, express it in the low-level representation of the database (e.g, MAL in case of MonetDB) and dispatch it directly to the execution engine for execution.

## 5.5 Pluggability

QFusor is built for portability, requiring minimal engine-specific adjustments. Its main dependency lies in parsing the query plan to construct a DFG (Section 5.1). While parsing raw SQL is possible, this would sacrifice the optimizer decisions and statistical insights available in the plan. Both fusion optimization and query rewriting are engine-agnostic; the rewritten queries are expressed in standard SQL. If necessary, adaptations to specific SQL dialects can be handled externally, using tools like SQLGlot [80].

JIT code generation (Section 5.3) includes engine-specific steps for code registration, which are handled as follows. The design specifications (Section 4.2) and CFFI wrappers enable pluggable integration with different SQL engines. Integration requires an engine-specific implementation of CREATE FUNCTION to register a C UDF (one per UDF type), which should invoke a standard wrapper function with the following prototype:

```
extern int udfwrapper(char* name, int n_inputs, InputData* inputs,
   int n_outputs, ResultData* outputs,
   char* extras, char** errormessage);
```

Each input or output is described using the following structure:

```
typedef struct {
   int type;      // 1: char**, 2: int*, 3: double*, etc.
   void* array;   // pointer to data array
   int size;      // number of elements
} InputData;
```

Pluggability across engines is facilitated by a `db_dialect.py` file, which defines engine-specific `CREATE FUNCTION` statements and data type mappings. Based on our experience with several data engines, this file typically contains 300–400 lines of Python code.

# 6 Experiments

## 6.1 Implementation and Deployment

QFusor implements a stateful UDF mechanism that wraps each Python UDF with CFFI (`v.1.14.6`) and then embeds it into a C UDF; C UDFs are supported by most databases. Then, UDFs are registered into the database with a `CREATE FUNCTION` statement, whose syntax is engine specific. QFusor has been prototyped on MonetDB, PostgreSQL, SQLite, DuckDB, PySpark, and a commercial database. We use PyPy (`v.7.3.6 with GCC 7.3.1`) as a tracing JIT compiler, which supports most popular Python packages.

## 6.2 Experimental Setup

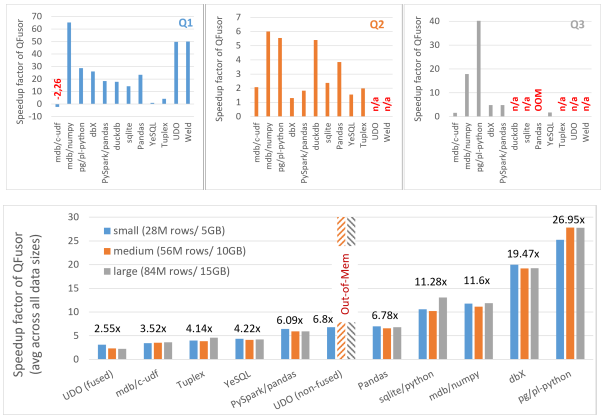
*6.2.1 Hardware and software.* All experiments ran on a server (Intel i7-4930K, 64GB memory) running Ubuntu 22.04. Unless otherwise stated, we report the average of 5 executions with cold caches on HDD disks. Software used: PostgreSQL (v.17.6), Pandas (v.1.3.5), Spark (PySpark, v.2.4.7), MonetDB (v11.44.0 - NumPy/C UDFs), SQLite (v. 3.31.1), DuckDB (v.1.3.2), NumPy (v1.21.5), CFFI (v1.13), nltk (v.3.6.7), and a commercial analytics database (dbX).

*6.2.2 Datasets and queries.* (a) `udfbench` is obtained from the UDFBench repository [25, 26, 83]. (b) `zillow` is obtained from Tuplex's repository and it is enhanced with aggregations and group-by's [79]. (c) `weld` comprises two queries introduced in the Weld paper [57]. (d) `udo` comprises two pipelines introduced in the UDO paper [72]. Figure 4 shows statistics of the queries used in the experiments.

*6.2.3 Methodology.* Our analysis has two parts. The first compares QFusor with state-of-the-art approaches such as Tuplex [79], UDO [71], Weld [57], YeSQL [27], MonetDB[50] with NumPy and C-UDF, Pandas [58], and data engines such as SQLite [82], PostgreSQL/pl-python [59], PySpark [63], and dbX. We investigate how performant various technologies are in running queries with UDFs. As discussed, Tuplex and Tupleware (not publicly available) do not employ a SQL database, but as they JIT compile end-to-end Python pipelines comprise and excellent baseline for our work. The second part presents micro experiments to test QFusor for scenarios including hot vs. cold caches, disk vs. memory, parallelization, compilation, resource utilization, etc. Unless otherwise stated, we present results based on our QFusor implementation on top of MonetDB.

## 6.3 Systems Comparisons

*6.3.1 UDFBench.* UDFBench [26] comprises 21 queries that use 42 scalar, aggregate, and table UDFs. The queries are grouped into four query classes (QC-1 to QC-4) based on their complexity. However, not all systems considered here support the full benchmark suite. A detailed compatibility matrix can be found in our artifacts repository [64]. For our evaluation, we selected three representative UDFBench queries supported by most systems: (`Q1`): UDFBench's Q1 (QC-1) is a simple query with 3 scalar UDFs and no beneficial fusion opportunity. (`Q2`): UDFBench's Q12 (QC-2) combines complex relational logic with scalar UDFs. (`Q3`): UDFBench's Q16 (QC-3) blends complex relational logic with complex UDFs. Q1 and Q2 are natively supported by all systems except UDO and Weld. For completeness, we adapted







| origin | udfbench | | | | | | | | | | zillow | | | | weld | | udo | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Q9 | Q10 | Q11 | Q12 | Q13 | Q14 | Q15 | Q16 | Q17 | Q18 |
| #udfs | 3 | 3 | 16 | 5 | 3 | 6 | 4 | 1 | 2 | 2 | 14 | 3 | 4 | 32 | 4 | 1 | 1 | 1 |
| #scalar | 3 | 2 | 12 | 2 | 2 | 5 | 4 | 1 | 2 | 2 | 9 | 3 | 3 | 26 | 2 | 0 | 0 | 0 |
| #aggr | 0 | 1 | 3 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 5 | 0 | 1 | 8 | 1 | 0 | 0 | 0 |
| #table | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| nest-dpth | 1 | 2 | 5 | 3 | 2 | 6 | 4 | 1 | 2 | 2 | 3 | 3 | 3 | 4 | 2 | 1 | 1 | 1 |
| #fused | 0 | 2 | 14 | 4 | 2 | 5 | 3 | 1 | 1 | 1 | 9 | 2 | 2 | 23 | 2 | 0 | 0 | 0 |
| fus-optim | 0.198 | 0.52 | 0.74 | 0.33 | 0.298 | 0.82 | 0.399 | 0.287 | 0.345 | 0.33 | 0.258 | 0.252 | 0.402 | 0.853 | 0.52 | 0.201 | 0.344 | 0.451 |
| code-gen | 0 | 0.2 | 0.57 | 0.295 | 0.374 | 0.345 | 0.205 | 0.345 | 0.382 | 0.396 | 0.173 | 0.441 | 0.355 | 0.966 | 0.391 | 0 | 0 | 0 |
| compil | 0 | 540 | 527 | 540.3 | 525 | 585.2 | 367.4 | 557.7 | 564 | 542.5 | 728.8 | 544 | 508.5 | 597.1 | 577.1 | 0 | 0 | 0 |
| runtime | 700 | 45234 | 121312 | 980 | 940 | 12100 | 16300 | 12211 | 2729 | 13212 | 61977 | 20169 | 1349 | 45198 | 16161 | 5495 | 23092 | 6156 |

**Figure 4: QFusor vs. SOTA: on `udfbench` (top) and `zillow` (middle) datasets, and the queries used in the experiments**

Q1 for both. In UDO, which only supports table UDFs, we implemented its scalar UDFs as table UDFs. In Weld, which supports NumPy-native operations rather than general Python UDFs, Q1 was rewritten into WeldIR. Q3 is supported natively only on a subset of the evaluated systems.

Figure 4(top) shows a comparison of QFusor with the other systems using the large UDFBench dataset [26]. Despite Q1 offers no fusion opportunities, QFusor still outperforms most engines due to its scalable UDF specifications and tracing JIT, performing on par with YeSQL that also executes Python UDFs on tracing JIT. MonetDB with C-UDF (mdb/c-udf) performs excellent by avoiding context switching through in-process execution in the same language as the engine. Weld and UDO show weaker performance in Q1, as they lack native support for UDFBench's scalar UDFs. In Q2, however, mdb/c-udf incurs overhead from intermediate materialization caused by operator-at-a-time execution. QFusor eliminates this cost via fusion optimization. SQLite's tuple-at-a-time model leads to numerous foreign function calls, while PostgreSQL introduces inter-process communication overhead by running UDFs in separate processes. By contrast, QFusor combines operator fusion with tracing JIT, avoiding these bottlenecks and achieving consistently superior performance. Pandas, DuckDB, PySpark with Pandas, and MonetDB with NumPy UDFs (mdb/numpy) also suffer from intermediate materialization. QFusor outperforms Tuplex, whose row-store layout forces unnecessary data processing. While YeSQL achieves good results through JIT compilation of fused scalar UDFs, QFusor extends fusion to relational operators yielding faster execution. Finally, dbX surpasses all engines except QFusor in Q2 due to strong parallelism, but its lack of UDF JIT compilation and context switches between relational and UDF operators limit performance. UDO and Weld do not support Q2 (n/a). In Q3, QFusor's integration with MonetDB outperforms the other engines from 2x (YeSQL) up to 40x (PostgreSQL) due to aggressive fusion of relational operators sum, case, and filter with the cleandate UDF and the fusion of the preprocessing UDF pipeline which consists of scalar and table UDFs. Q3 is not supported (n/a) in all systems.

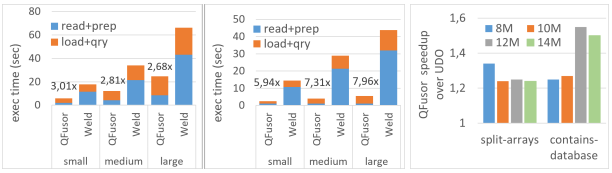



**Figure 5: sys vs. Weld: `get_population_stats` (left) and `data_cleaning` (middle), and QFusor vs. UDO (right)**

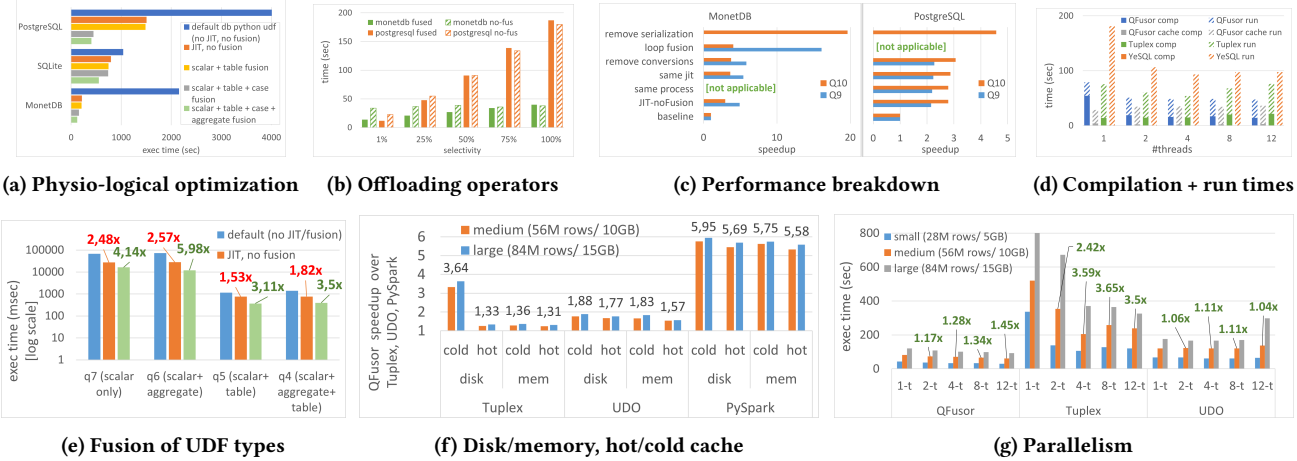
*6.3.2 Zillow dataset.* Figure 4(middle) shows results for the Zillow pipeline (Q11 query), which mainly involves string operations. QFusor clearly outperforms all other systems. While numpy, pandas, and mdb/numpy perform well with numeric data, they lack support of complex string operations, which are executed by CPython's interpreter leading to slower execution. PostgreSQL, UDO, Sqlite, and MonetDB do not employ fusion and suffer from data copies, conversions, materializations, and function call overheads due to their tuple-at-a-time execution model. These issues are exacerbated with string data due to its higher conversion cost. We tested two variants of UDO: non-fused (the default) and manually fused by us. The non-fused failed on medium (10GB) and large (15GB) datasets, requiring 96GB of memory for the medium case (our server has 64GB) showing that it is particularly memory demanding (see also Figure 7). The manually fused version run efficiently with less memory (18GB), showing the effectiveness of operator fusion when paired with UDO's C++ UDFs. tuplex supports parallelism via data partitioning, but this introduces overhead compared to MonetDB, which parallelizes operators without partitioning. YeSQL fuses only scalar UDFs, which is not sufficient in this pipeline. Note that Q11 includes relational operators (e.g., filters, group by's, native aggregations) executed before or after UDFs. QFusor effectively fuses these operators and offloads the relational operators in the UDF environment, delivering substantial performance gains in this experiment.

*6.3.3 QFusor vs. Weld.* As Weld does not support natively UDFBench, for fairness, we compared it with QFusor, using two Weld queries [57]: (Q15) `get_population_stats` and (Q16) `data_cleaning`, on their respective datasets across three sizes: small (7M/7.5M rows), medium (15M/15M rows), and large (30M/22M rows). Figure 5 shows that QFusor outperforms Weld, showing with hot caches an average speedup in total compute time 2.83x and 7x for `get_population_stats` and `data_cleaning`, respectively. Note that Weld loads data in two phases. In the first phase, an external CSV file is read and parsed (preprocessing) into a Python dataframe. In the second phase, the data is loaded into the Weld runtime during execution. In contrast, QFusor follows a different architecture, with corresponding phases labeled read and execute.

*6.3.4 QFusor vs. UDO.* For fairness, we compared QFusor with UDO using two pipelines from UDO's repo: (Q17) `split arrays` and (Q18) `contains-database`. These pipelines have no fusion opportunities. Hence, we measure the performance of our JIT compiled execution vs. the out-of-the box UDO execution. Figure 5(right) shows that with hot caches, on average QFusor is 27% and 39% faster than UDO.

## 6.4 Scrutinizing QFusor

*6.4.1 Physio-logical optimization.* We use Q3 (our running example) on the large UDFBench dataset and compare QFusor integrated with MonetDB, PostgreSQL, and SQLite (see Figure 6a).




**(a) Physio-logical optimization** **(b) Offloading operators** **(c) Performance breakdown** **(d) Compilation + run times**

**(e) Fusion of UDF types** **(f) Disk/memory, hot/cold cache** **(g) Parallelism**

**Figure 6: Scrutinizing QFusor**

We evaluate five techniques: (a) default Python UDF execution (`no fusion/JIT`); (b) JIT only (`JIT, no fusion`); (c) add fusion of `scalar+table` functions; (d) add offloading of scalar relational operators and operator reordering (`case`, `filter`); and (e) add offloading of `aggregations` (`sum`, `groupby`), with exporting internal functions in MonetDB (Section 5.3.2). After applying these techniques, we create the fused query of Figure 2. Different execution models (`PostgreSQL`'s out-of-process UDFs, `SQLite`'s in-process UDFs, `MonetDB`'s in-process vectorized UDFs) and different reordering of the UDF `cleandate` account for performance differences among the databases. Specifically, `PostgreSQL` does not push down the UDF `cleandate`, resulting in 3x more UDF invocations than MonetDB. In contrast, `SQLite` stalls under the native CPython execution and completes in 1042s only when the query is decomposed into two parts using `create temp table`. Still, QFusor delivers up to a 18x speedup, demonstrating its applicability and effectiveness across database systems.

*6.4.2 Operator offloading.* To study further the impact of offloading relational operators to the UDF environment, we crafted query `Q8` and ran it on the large UDFBench dataset [26] with varying selectivities (1%–100%). `Q8` is based on a subexpression of `Q2` applying the `cleandate` UDF before a `range filter`. Figure 6b shows that in `MonetDB`, non-fused (`no-fus`) JIT execution presents constant runtime due to the overhead of copying UDF results back to the engine. Fusing the filter with the UDF in the UDF runtime avoids unnecessary materialization for low selectivity, yielding up to 2.4x speedup. For high selectivity, fusion yields diminishing returns. `PostgreSQL` starts faster for low selectivity due to parallelism, and benefits from filter offloading due to reduced UDF output materialization. But for high selectivity, the inter-process communication takes a toll.

*6.4.3 Physical optimization.* To measure the impact of physical optimizations, we crafted two queries `Q9` and `Q10` based on `udfbench` (large). `Q9` uses two UDFs (`cleandate`, `extractmonth`) from UDFBench to expose compilation overheads. `Q10` involves complex data structures (Python lists serialized using json) and two UDFBench UDFs: (a) `jpack`: tokenizes an input string and packs it in a json array, and (b) `jsoncount`: parses the json array and counts the tokens. We test the following techniques on `MonetDB` (column store, operator-at-a-time) and `PostgreSQL` (row store, tuple-at-atime): (a) `Baseline`: native Python UDFs in `MonetDB` and `PostgreSQL`. (b) `JIT-noFusion`: QFusor with tracing JIT UDFs and fusion disabled. (c) `Same process`: run UDFs in-process; default in `MonetDB`, in `PostgreSQL`, the UDFs are called from the same C UDF. (d) `Same JIT`: run the UDFs in the same tracing JIT execution trace, with a wrapper Python function that calls the UDFs sequentially. (e) `Remove C↔JIT conversions`: skip data conversions by passing the output of one UDF directly to the second using a wrapping function. (f) `Loop fusion` [70]: eliminate intermediate materializations; in `MonetDB`, a wrapper function calls the UDFs sequentially in a single loop, whereas in `PostgreSQL`, pipeline execution is the default operation. (g) `Remove serialization`: eliminate serialization overheads in fusion code generation.

Figure 6c shows that all techniques yield performance improvements, leading to an overall speedup of 20x in `MonetDB` and 4.6x in `PostgreSQL`. In `Q10`, `PostgreSQL`'s native performance (35s) surpasses `MonetDBs` (53s) as it executes the more demanding Python UDFs (involving JSON serialization) in separate processes, thereby avoids acquiring the GIL. However, in `Q9` that applies lightweight UDFs over a large table, `Postgres` (385s) suffers from inter-process communication overheads and runs slower than `MonetDB` (212s). After applying our optimizations, `MonetDB` (Q9: 13.2, Q10: 2.7s) accelerates query execution more aggressively than PostgreSQL (Q9: 169s, Q10: 7.6s). This is due to MonetDB's vectorized execution and our techniques that enable longer JIT traces and eliminate intermediate results through loop fusion. Our method for handling complex data types (Section 4.2) greatly reduces serialization overhead, leading to substantial performance gains for `Q10` across both engines.

*6.4.4 Fusion optimization overhead.* Figure 4 (bottom) shows the overhead (in msec) of the QFusor steps: `fus-optim` = finding fusible operators + fusion optimization, and `code-gen` = query and fused UDF code generation, for all queries used in the experiments. As we observe, these overheads do not affect much query runtime.

*6.4.5 Fusion compilation latency.* We study the efficiency of fusion compilation across three cases: QFusor with tracing JIT, `tuplex` with LLVM, and `YeSQL` without fusion that performs no online compilation. For compatibility with `tuplex`, here we use the `zillow` dataset. To isolate and highlight compilation overhead, which is particularly relevant in short-lived queries, we run two queries, a small one (`Q13`) and a complex one (`Q14`), on a

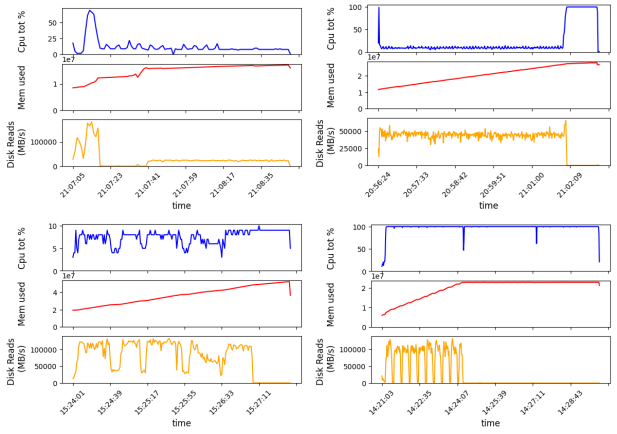



**Figure 7: QFusor (top-left), Tuplex (top-right), UDO (bottom-left), PySpark (bottom-right): resource utilization**

tiny zillow snapshot (785K rows/ 0.14GB). QFusor compilation and optimization lasts 510ms and 520ms with runtime 1.349s and 2.573s, for Q13 and Q14, respectively. tuplex compilation is 396ms and 2.48s with runtime 2.647s and 8.111s. YeSQL's runtime is 1.668s and 6.676s. Hence, the QFusor compilation overhead remains low regardless of query complexity whilst LLVM gets more expensive for complex queries.

Next, we investigate fusion gains and latency of QFusor in a workload of 100 short-running queries involving scalar and aggregate UDFs (variants of Q11, Q12, Q13, and Q14) on tiny zillow. We compare QFusor, tuplex, and YeSQL, with varying parallelism. Figure 6d shows that QFusor executes the workload faster than tuplex, with fusion benefits outweighing compilation latency, with the expected exception of single threaded execution. QFusor cache is a variant that caches previously compiled UDFs and examines potential re-use benefits (zero compilation cost) in future queries. The results show an excellent potential for such an approach.

*6.4.6 UDF types.* We investigate the effect of fusing various UDF types using queries Q4, Q5, Q6, and Q7 on the artifacts table of UDF-Bench (large) [26]. Figure 6e presents the results with hot caches. QFusor achieves speedups up to 6x in all cases, providing further evidence that UDF loop fusion effectively reduces unnecessary overhead. For brevity, we omit results with cold caches; however, the trends are consistent with those observed for hot caches.

*6.4.7 Disk vs. main-memory.* We compare QFusor vs. tuplex, UDO, and PySpark on zillow (Q2 query) stored on disk and in main memory. Figure 6f shows execution times for cold and hot caches. QFusor outperforms PySpark by 5.75x and UDO (manually fused query) by 1.76x, on average, regardless of storage or caching mode. When reading from disk-based tables, QFusor is 3.64x faster than tuplex, which reads from disk-based CSV files. Clearly, the read/load phase of tuplex takes a toll. With hot caches or in-memory data, we measure only the compute part of tuplex, which is ~1.33x slower than QFusor. Given that tuplex was expected to provide the best possible, out-of-database Python execution due to its end-to-end JIT compilation, this is an excellent result. The QFusor speedup is justified by (a) the benefit of query optimization (tuplex does not exploit this) as QFusor works in synergy with a database, and (b) the faster compilation method, as we discussed in Section 6.4.5.

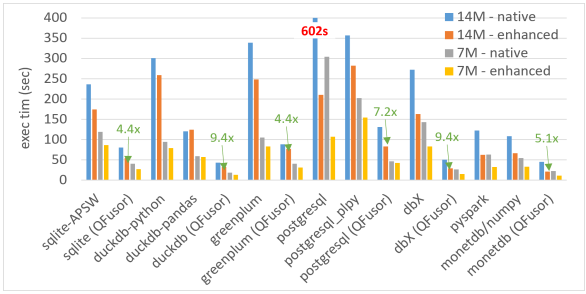



**Figure 8: QFusor pluggability to data engines**

*6.4.8 Parallelism.* Figure 6g shows how QFusor, tuplex, and UDO scale with 1 to 12 pthreads on zillow (Q2 query). QFusor is faster than tuplex and UDO. Its performance increases with more threads and achieves a 45% speedup for 12 threads. Multi-threaded parallelism is limited by Python's GIL and is more effective for relational operators run in the database. tuplex has speedups up to 3.6x for 4 threads. However, when utilizing more threads, data partitioning adds significant overheads and its performance reaches a plateau. UDO seem to reap marginal benefit from multi-threaded execution.

*6.4.9 Resource utilization.* We compare resource usage (CPU, memory, disk I/O) of QFusor with tuplex, UDO, and PySpark using the Q2 query on the large zillow data in multi-threaded execution (Figure 7). QFusor finishes in 92sec with CPU usage mainly below 20% (due to Python's GIL), moderate memory usage up to 2GBs, and fast loading (11sec). Notably, it starts query processing with data loading. tuplex runs in 378sec, uses 1-3GB of memory and exhibits sustained I/O activity. It also overlaps data loading with query processing, and as an optimization it generates a CSV parser that inlines the function logic [79]. UDO runs in 190sec with low CPU usage, high disk I/O, and aggressive use of memory (2-4GB). PySpark runs in 460sec with high CPU usage and memory up to 2GBs.

*6.4.10 Pluggability to other engines.* We test QFusor integration with various data engines, using Q12 that comprises 3 UDFs on the url column of zillow for 7M and 14M rows. On each engine, we test two modes: native (run Q12 as is) and enhanced (run a hand-crafted fused version of Q12)). On engines integrated with QFusor, JIT compilation is always on, and the two modes indicate whether fusion is 'off' (native) or 'on' (enhanced). Figure 8 shows the results of the test, and also shows the speedup of QFusor (on 14M rows). The benefit of QFusor is evident across engines.

## 7 Conclusions

We presented QFusor, a novel stateful system that enables the fusion of UDFs and relational operators within SQL databases. Supporting scalar, aggregate, and table UDFs, QFusor leverages a tracing JIT compiler to significantly accelerate UDF execution. Its optimizer complements the database's native query optimizer, dynamically generating efficient execution plans at runtime. Designed to be easily integrated with a range of popular databases, QFusor delivers substantial performance improvements, achieving up to 40x speedup over state-of-the-art alternatives.

**Acknowledgments.** This work was partially supported by the EU Horizon Europe programmes: DataGEMS (GA.101188416), CREX-DATA (GA.101092749), and EBRAINS 2.0 (GA.101147319). We thank our colleague Theoni Palaiologou for her assistance with PostgreSQL experiments.

## Artifacts

The datasets and the queries used in the experiments can be found here: https://github.com/athenarc/QFusor.

## References


[1] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *USENIX NSDI 17*. 469–482.

[2] Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann.

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

[4] Samuel Arch, Yuchen Liu, Todd C. Mowry, Jignesh M. Patel, and Andrew Pavlo. 2024. The Key to Effective UDF Optimization: Before Inlining, First Perform Outlining. *Proc. VLDB Endow.* 18, 1 (2024), 1–13.

[5] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcín, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Comput. Sci. Eng.* 13, 2 (2011), 31–39.

[6] A. J. Bernstein. 1966. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers* EC-15, 5 (1966), 757–763.

[7] 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*.

[8] 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. *Proc. VLDB Endow.* 11, 12 (2018), 1755–1768.

[9] Michael J. Cafarella and Christopher Ré. 2010. Manimal: Relational Optimization for Data-Intensive Programs. In *WebDB*.

[10] CFFI. 2022. Using the ffi/lib objects. Available at: https://cffi.readthedocs.io/en/latest/using.html.

[11] Surajit Chaudhuri and Kyuseok Shim. 1993. Query Optimization in the Presence of Foreign Functions. In *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, Rakesh Agrawal, Seán Baker, and David A. Bell (Eds.). Morgan Kaufmann, 529–542.

[12] Surajit Chaudhuri and Kyuseok Shim. 1996. Optimization of Queries with User-defined Predicates. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda (Eds.). Morgan Kaufmann, 87–98.

[13] Surajit Chaudhuri and Kyuseok Shim. 1999. Optimization of Queries with User-Defined Predicates. *ACM Trans. Database Syst.* 24, 2 (1999), 177–228.

[14] Hanfeng Chen, Joseph Vinish D'silva, Hongji Chen, Bettina Kemme, and Laurie Hendren. 2018. HorseIR: Bringing Array Programming Languages Together with Database Query Processing. In *ACM SIGPLAN DLS*. 37–49.

[15] 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.

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

[17] 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.

[18] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. 2015. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*. www.cidrdb.org.

[19] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. 2018. AIDA - Abstraction for Advanced In-Database Analytics. *PVLDB* 11, 11 (2018), 1400–1413.

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

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

[22] 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.

[23] 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*. USENIX Association, 799–815.

[24] Tim Fischer, Denis Hirn, and Torsten Grust. 2024. SQL Engines Excel at the Execution of Imperative Programs. *Proc. VLDB Endow.* 17, 13 (2024), 4696–4708.

[25] Yannis Foufoulas, Theoni Palaiologou, and Alkis Simitsis. 2025. UDFBench: A Tool for Benchmarking UDF Queries on SQL Engines. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 91–94.

[26] Yannis Foufoulas, Theoni Palaiologou, and Alkis Simitsis. 2025. The UDFBench Benchmark for General-purpose UDF Queries. *Proc. VLDB Endow.* 18, 9 (2025), 2804–2817.

[27] 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), 2270–2283.

[28] Antoine Fraboulet, Karen Kodary, and Anne Mignotte. 2001. Loop fusion for memory space optimization. In *Proceedings of the 14th international symposium on Systems synthesis*. 95–100.

[29] Kai Franz, Samuel Arch, Denis Hirn, Torsten Grust, Todd C. Mowry, and Andrew Pavlo. 2024. Dear User-Defined Functions, Inlining isn't working out so great for us. Let's try batching to make our relationship work. Sincerely, SQL. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org.

[30] Henning Funke, Jan Mühlig, and Jens Teubner. 2022. Low-latency query compilation. *VLDB J.* 31, 6 (2022), 1171–1184.

[31] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. 2021. Babelfish: Efficient execution of polyglot queries. *Proceedings of the VLDB Endowment* 15, 2 (2021), 196–210.

[32] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *SIGMOD*. 559–573.

[33] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding their usage in the wild. *Proc. VLDB Endow.* 14, 8 (2021), 1378–1391.

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

[35] Joseph V. D'silva Hanfeng Chen, Laurie Hendren, and Bettina Kemme. 2021. HorsePower: Accelerating Database Queries for Advanced Data Analytics. In *Proceedings of the 24th International Conference on Extending Database Technology (EDBT)*. EDBT, 361–366.

[36] Joseph M. Hellerstein. 1994. Practical Predicate Placement. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, Richard T. Snodgrass and Marianne Winslett (Eds.). ACM Press, 325–335.

[37] Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate Migration: Optimizing Queries with Expensive Predicates. In *SIGMOD*. 267–276.

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

[39] 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*. 1292–1295.

[40] Fabian Hueske, Mathias Peters, Matthias Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. 2012. Opening the Black Boxes in Data Flow Optimization. *Proc. VLDB Endow.* 5, 11 (2012), 1256–1267.

[41] 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*.

[42] Michael Jungmair, Alexis Engelke, and Jana Giceva. 2024. HiPy: Extracting High-Level Semantics from Python Code for Data Processing. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 736–762.

[43] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* 15, 11 (2022), 2389–2401.

[44] 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), 2209–2222.

[45] Steffen Kläbe, Bobby DeSantis, Stefan Hagedorn, and Kai-Uwe Sattler. 2022. Accelerating Python UDFs in Vectorized Query Execution. In *Proceedings of the Annual Conference on Innovative Data Systems Research*. CIDR.

[46] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *LLVM-SC*.

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

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

[49] Microsoft. 2023. Language Integrated Query (LINQ). Available at: https://learn.microsoft.com/en-us/dotnet/csharp/linq.

[50] MonetDB. 2022. Available at: https://www.monetdb.org.

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

[52] Thomas Neumann. 2021. Evolution of a Compiling Query Engine. *Proc. VLDB Endow.* 14, 12 (2021), 3207–3210.

[53] Nuitka the Python Compiler. 2022. Available at: https://nuitka.net.

[54] OpenAire. 2023. Available at: https://www.openaire.eu/.

[55] OpenAire. 2023. Information Inference Service (IIS). Available at: https://github.com/openaire/iis.

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

[57] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating end-to-end optimization for data analytics applications in weld. In *Proceedings*

*of the VLDB Endowment*. VLDB, 1002–1015.
[58] Pandas. 2022. Available at: https://pandas.pydata.org.
[59] PostgreSQL. 2022. Available at: https://www.postgresql.org.
[60] PostgreSQL. 2022. PL/pgSQL, SQL Procedural Language. Available at: https://www.postgresql.org/docs/current/plpgsql.html.
[61] PostgreSQL. 2022. PostgreSQL, PL/Python Functions Available at: https://www.postgresql.org/docs/current/plpython-funcs.html.
[62] PostgreSQL. 2022. SQL commands, CREATE FUNCTION. Available at: https://www.postgresql.org/docs/current/sql-createfunction.html.
[63] PySpark. 2022. Available at: https://www.databricks.com/glossary/pyspark.
[64] QFusor. 2025. Artifacts repository for QFusor. Available at: https://github.com/athenarc/QFusor.
[65] Mark Raasveld and Hannes Mühleisen. 2016. Vectorized UDFs in Column-Stores. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*. SSDBM, 1–12.
[66] 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.
[67] 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.
[68] Hesam Shahrokhi, Callum Groeger, Yizhuo Yang, and Amir Shaikhha. 2023. Efficient Query Processing in Python Using Compilation. In *SIGMOD*. 199–202.
[69] Hesam Shahrokhi and Amir Shaikhha. 2023. Building a Compiled Query Engine in Python. In *SIGPLAN*. ACM, 180–190.
[70] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. 2018. Push versus pull-based loop fusion in query engines. *J. Funct. Program.* 28 (2018), e10.
[71] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *PVLDB* 15, 5 (2022), 1119–1131.
[72] Moritz Sichert and Thomas Neumann. 2022. User-defined operators: efficiently integrating custom algorithms into modern databases. *Proceedings of the VLDB Endowment* 15, 5 (2022), 1119–1131.
[73] Tarique Siddiqui, Arnd Christian König, Jiashen Cao, Cong Yan, and Shuvendu K. Lahiri. 2025. QURE: AI-Assisted and Automatically Verified UDF Inlining. *Proc. ACM Manag. Data* 3, 1 (2025), 66:1–66:26.
[74] Varun Simhadri, Karthik Ramachandra, Arun Chaitanya, Ravindra Guravannavar, and S. Sudarshan. 2014. Decorrelation of user defined function invocations in queries. In *ICDE*. 532–543.
[75] Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. 2005. Optimizing ETL Processes in Data Warehouses. In *ICDE*. 564–575.
[76] Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. 2005. State-Space Optimization of ETL Workflows. *IEEE Trans. Knowl. Data Eng.* 17, 10 (2005), 1404–1419.
[77] Alkis Simitsis, Panos Vassiliadis, Manolis Terrovitis, and Spiros Skiadopoulos. 2005. Graph-Based Modeling of ETL Activities with Multi-level Transformations and Updates. In *7th International Conference on Data Warehousing and Knowledge Discovery, DaWaK, Copenhagen, Denmark, August 22-26, 2005 (Lecture Notes in Computer Science, Vol. 3589)*. Springer, 43–52.
[78] Spark. 2023. Available at: https://spark.apache.org.
[79] Leonhard F Spiegelberg, Rahul Yesantharao, Malt Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD, 1718–1731.
[80] SQLGlot. 2025. Python SQL Parser and Transpiler. Available at: https://sqlglot.com/sqlglot.html.
[81] SQLITE. 2022. Application-Defined SQL Functions Available at: https://www.sqlite.org/appfunc.html.
[82] SQLITE. 2022. Available at: https://www.sqlite.org.
[83] UDFBench. 2025. Code repository. Available at: https://github.com/athenarc/UDFBench.
[84] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. 2009. Symbolic Query Exploration. In *ICFEM*, Vol. 5885. 49–68.
[85] Vertica. 2022. Extending Vertica. Available at: https://www.vertica.com/docs/12.0.x/HTML/Content/Authoring/ExtendingVertica/ExtendingVertica.htm.
[86] Johannes Wehrstein, Tiemo Bang, Roman Heinrich, and Carsten Binnig. 2025. GRACEFUL: A Learned Cost Estimator For UDFs. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong SAR, China, May 19 – 23, 2025*. IEEE.
[87] Cong Yan, Yin Lin, and Yeye He. 2023. Predicate Pushdown for Data Science Pipelines. *Proc. ACM Manag. Data* 1, 2 (2023), 136:1–136:28.
[88] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *USENIX*. 1–14.