



User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases

Moritz Sichert

Technische Universität München
moritz.sichert@tum.de

Thomas Neumann

Technische Universität München
neumann@in.tum.de

ABSTRACT

In recent years, complex data mining and machine learning algorithms have become more common in data analytics. Several specialized systems exist to evaluate these algorithms on ever-growing data sets, which are built to efficiently execute different types of complex analytics queries.

However, using these various systems comes at a price. Moving data out of traditional database systems is often slow as it requires exporting and importing data, which is typically performed using the relatively inefficient CSV format. Additionally, database systems usually offer strong ACID guarantees, which are lost when adding new, external systems. This disadvantage can be detrimental to the consistency of the results.

Most data scientists still prefer not to use classical database systems for data analytics. The main reason why RDBMS are not used is that SQL is difficult to work with due to its declarative and set-oriented nature, and is not easily extensible.

We present User-Defined Operators (UDOs) as a concept to include custom algorithms into modern query engines. Users can write idiomatic code in the programming language of their choice, which is then directly integrated into existing database systems. We show that our implementation can compete with specialized tools and existing query engines while retaining all beneficial properties of the database system.

PVLDB Reference Format:

Moritz Sichert and Thomas Neumann. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. PVLDB, 15(5): 1119 - 1131, 2022.
doi:10.14778/3510397.3510408

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/tum-db/user-defined-operators>.

1 INTRODUCTION

Modern data analytics has evolved to include complex algorithms for data mining and machine learning. Specialized systems have been designed that are able to handle ever-growing amounts of data to solve a larger variety of problems. Unfortunately, traditional systems, especially RDBMS, seem difficult to adapt to state-of-the-art data analytics [28].

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 15, No. 5 ISSN 2150-8097.
doi:10.14778/3510397.3510408

User-Defined Operators

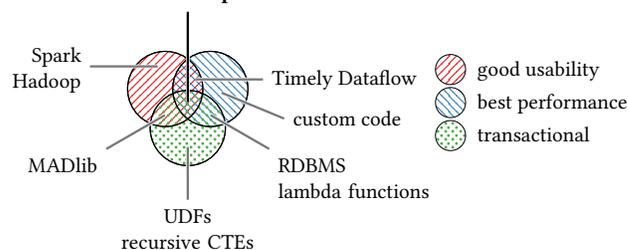


Figure 1: Comparison of approaches for modern data analytics. Systems with better usability usually have poor performance. User-Defined Operators can combine all three properties.

Still, most data that is eventually analyzed in special-purpose systems is originally sourced from the RDBMS. Thus, a common approach is to create ETL workflows that can accommodate the use of different systems [31]. ETL workflows usually collect data in a data warehouse, which is built on top of the RDBMS that supports SQL. Next, the data is exported to be further processed by the data analytics systems. This extraction process can be implemented naively by exporting to a CSV file from the data warehouse and then importing the data into the analytics systems. Additionally, some systems support more efficient data transfer by directly communicating with the RDBMS. As the last step, the results of the analysis are often transferred back to the data warehouse so that they can be further processed, for example, the data displayed in a dashboard. Exporting and importing data can often consume a considerable amount of time, especially when the data must be serialized to and parsed from a text format like CSV.

When data is extracted from an RDBMS, many beneficial characteristics of the system are lost. In particular, ACID properties can no longer be guaranteed. One could argue that atomicity and isolation may be of minor importance for systems that mainly process read-only OLAP workloads. However, to provide near real-time data analytics, data warehouses must be periodically updated. In combination with longer-running analytics queries, this means that the system must support proper transaction management to provide accurate results.

An additional benefit of modern database systems such as Umbra [23] or DuckDB [26] is their execution speed. The modern database systems use techniques such as code-generation [22] or vectorized execution [6], to be able to saturate the memory bandwidth, thereby processing analytical queries very quickly.

In practice, the advantages of using RDBMS do not outweigh the disadvantages. Data scientists often prefer using systems such as Spark [29], TensorFlow [4], or systems using the MapReduce

paradigm [8], even if they are not able to reach main-memory performance. In these systems, algorithms can be formulated in procedural programming languages such as Java, Scala, or Python. In comparison to SQL – the primary query language for RDBMS – those languages are better suited to formulate algorithms used in modern data analytics. Such algorithms are often iterative and can be expressed naturally by using code that contains assignments to variables and control-flow statements such as loops.

SQL, however, is declarative, set-oriented, and in general very different from most programming languages. Because of these differences, SQL provides query engines with a lot of flexibility in deciding how to execute a query. Query engines of RDBMS conceptually process streams of tuples. Each algebraic operator generates a stream of tuples and takes the streams of any number of input operators. The feature set of SQL is closely tied to this concept of stream processing, which supports filtering, several join types, and a wide range of aggregation and window functions. However, adding new functions is not easily possible. The execution of SQL queries is tightly coupled to the specific query engine of each database system, so adding new features to SQL requires deep knowledge of database internals.

Database systems do offer some extensibility of SQL. Imperative control flow can be formulated in SQL by using recursive CTEs [9, 14]. Also, some RDBMS offer imperative extensions to SQL such as T-SQL in Microsoft SQL Server [3] or PL/pgSQL in Postgres [1]. Imperative extensions allow users to write User-Defined Functions (UDFs), which can be called from standard SQL queries. The query engine can directly execute queries containing calls to UDFs written in this extended language. In theory, this functionality makes it much easier to write more complex algorithms when compared to recursive CTEs. However, the execution of functions written in these imperative languages tends to be very slow [12].

Additionally, UDFs are usually not allowed to take streams as arguments or return them. Instead, UDFs are applied to each tuple of one particular stream, which is similar to how a map operation works in other systems. Clearly, algorithms used in data analytics are not often expressed only by map-like operations. Thus, UDFs are not suitable to efficiently integrate custom algorithms in database systems.

To summarize, we identify the following problems with modern data analytics systems:

- (1) Processing data is inefficient due to costly import and export processes between different systems. Additionally, many systems cannot use modern hardware to its full capacity.
- (2) The most efficient systems have poor usability, and use SQL as a query language that is difficult to extend for data analytics.
- (3) Data may not be consistent; especially when extracting from an RDBMS, ACID properties can no longer be guaranteed.

Figure 1 shows an overview of the different approaches for modern data analytics. SQL-based approaches can usually guarantee consistency of the data but do not allow custom algorithms to be easily implemented. Analytics frameworks like Spark or Timely Dataflow [20] can automatically scale up user-written code but often do not reach the throughput of custom-written code or modern RDBMS with main-memory speeds.

In this work, we present the concept and implementation of *User-Defined Operators* (UDOs). Our approach solves all three problems with modern data analytics systems and can unify many beneficial properties in one system. As data is processed directly in the existing RDBMS, the data can be analyzed very efficiently within ACID transactions. In addition, a simple API allows users to write code in their language of choice.

This paper is structured as follows: In Section 2, we discuss related work in data analytics. In Section 3, we present our general concept of UDOs. Section 4 describes our implementation of UDOs in the code-generating database system Umbra [23] and Section 5 describes our implementation in Postgres. We present our evaluation in Section 6 and conclude our work in Section 7.

2 RELATED WORK

As mentioned above, imperative extensions to SQL can be used to write UDFs, which enables easier implementation of iterative algorithms and also allows the users to reuse existing database functionality. Gupta et al. [12] show that the extensions tend to be slow and cannot reach main-memory speeds. With their findings they encourage more research to improve the execution of UDFs.

A very popular approach for large-scale data analytics is MapReduce, which is often used for Big Data analytics but can also be applied to UDFs. Friedman et al. present an approach to formulate UDFs in MapReduce and integrate them into SQL queries [10].

Furthermore, separating UDFs into calls to a few predefined functions like map and reduce can be used to compile UDFs in the database and integrate them into the query execution as shown by Crotty et al. [7] who extend the MapReduce concept by adding more functions such as selection, join, or loop so that UDFs can address more different use cases. To execute this functionality efficiently, they make use of the LLVM framework [17] to be able to apply low-level optimization techniques to the generated queries.

Zou et al. [32] present an optimized approach to automatically partition workloads that contain UDFs. This approach allows users to develop highly scalable data analytics pipelines without in-depth knowledge of writing scalable code.

Palkar et al. [24] propose the Weld framework for data analytics, which combines code from different systems, such as queries written in SQL and programs written in Python. This framework uses a novel intermediate representation that can be lowered to LLVM.

Timely Dataflow is a novel concept for scalable data analytics presented in the Naiad system by Murray et al. [20]. This concept provides a low-level interface to assemble computational graphs, upon which high-level libraries and applications can be built. Murray et al. also present an implementation in the Rust programming language [21].

Writing code manually for a specific problem can lead to rapid execution but may not be easily combined with existing modern database systems. Passing et al. [25] present *lambda functions*, which are used to customize specific code-generating operators with greater ease. Schüle et al. [27] show how this technique can be applied to just-in-time compilation of user-written lambda functions in Postgres.

MADlib is a library of data analytics presented by Hellerstein et al. [13], which can extend database systems such as Postgres, and contains several algorithms which can be used directly from SQL.

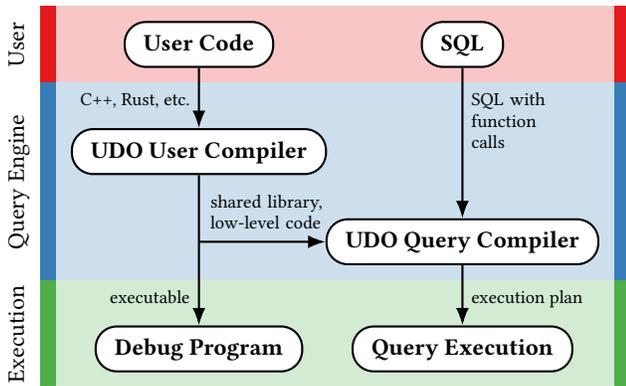


Figure 2: Architecture of the UDO User Compiler and UDO Query Compiler

Techniques used in modern main-memory databases such as code generation and vectorized execution can also be used for data analytics. Zhang et al. [30] present a system that automatically analyzes code to dynamically generate optimal query plans at runtime.

Duta et al. and Hirn et al. [9, 14] describe an approach that allows users to write code in the procedural programming language PL/SQL. Such PL/SQL programs are transformed entirely to recursive CTEs. This technique allows procedural programs to be interpreted by any SQL engine that supports recursive CTEs, and works well for many use cases. The result of the transformation is standard SQL, which can theoretically be executed in a main-memory database, provided it has an efficient implementation of recursive CTEs.

3 THE USER-DEFINED OPERATOR

To achieve good usability and performance while maintaining ACID properties, we present the novel *User-Defined Operator* (UDO), which represents user-written algorithms as *algebraic operators*, which extend the relational algebra utilized by existing RDBMS. UDOS solve the problems in data analytics mentioned in Section 1 as follows:

Usability: Even though UDOS are deeply integrated into query and execution engines of existing RDBMS as algebraic operators, users can use a simple API in the programming language of their choice. The complexity of writing efficient query engines is entirely hidden, and no knowledge of database internals is required. As query engines treat UDOS as regular algebraic operators, they can process arbitrary streams of tuples to generate a new output stream. This approach also interacts nicely with SQL: The input streams given to an UDO can be the result of arbitrary SQL queries containing joins and aggregations. Similarly, the output of the UDO can be further processed by using SQL.

Performance: Our implementation can optimize queries containing UDOS very efficiently. When those queries are executed in our code-generating database system Umbra, we can generate code that is as efficient as complex native operators written by database experts. As UDOS are directly executed in the database, the costly exporting and importing of data is not required.

Consistency: As UDOS are directly integrated into the query engine of existing RDBMS, they preserve all the ACID properties guaranteed by the system. User-written code is not required to take any precautions regarding consistency or isolation; the tuple streams utilized as input by UDOS follow the standard semantics of the isolation levels in SQL.

3.1 Overview

Figure 2 shows the overview of our architecture to compile and integrate UDOS into a query engine. The user writes a standard SQL query, and the query uses the UDO given by the user as source code of an imperative programming language, such as C++ or Rust.

First, the UDO code is processed and analyzed to detect errors in the code. Next, the UDO must be translated into a representation that the query engine can use. We call this part of the system the *UDO User Compiler*; as an additional feature it can directly generate a debug program that allows flexible debugging of the user-code independent of the database system.

In a second step, the query engine takes the processed user code and integrates it into the query plan that is generated from the SQL query. This part of the system is called the *UDO Query Compiler*, which implements the representation of the UDO as an algebraic operator.

Both compilers can be developed separately. Once a suitable API for user-written code is defined, several implementations for different query execution models can be developed. Different UDO User Compilers can accept code written in different programming languages, while the UDO Query Compiler will usually be tightly connected to the rest of the database system.

In the following, we explain the concepts of an UDO with the help of the following example. A blog website uses a relational database system to store its blog posts. The writer of the blog wants to analyze the posts by category, and is interested in how many blog posts of the category “lifestyle” were written and how many posts of the other categories exist. Listing 1 shows a query that uses a UDO written in C++ that can answer the writer’s questions. The query uses existing SQL syntax; the create function statement creates a new function with the name `count_lifestyle` (C), which takes a table as an input that must match the schema specified by `InputTuple` (S). Thus, any subquery with a string attribute with the name “word” can be used as an input to this UDO. The function also returns a table, which means that it can be used like a relation in the from part of a SQL query (F). Because the language is set to UDO-C++, the SQL parser knows that this function is an UDO written in C++. The user-written code is included in the SQL statement (1) – (3) and will eventually be processed by the UDO User Compiler and UDO Query Compiler.

3.2 The UDO User Compiler

In this section we introduce the *UDO User Compiler*. We define a high-level API that allows user-written code to be used by the User-Defined Operator. This API consists of only three functions, and it is not specific to any particular programming language. Our implementation provides a C++ API (refer to Section 4) but it is general enough to be implemented for most programming languages.

```

create function count_lifestyle(table) ③
returns table language 'UDO-C++' as $$
struct InputTuple { udo::String word; }; ⑤
struct OutputTuple { udo::String word; uint64_t n; };
class CountLifestyle
: public UDOoperator<InputTuple, OuputTuple> {
    uint64_t lifestyle = 0, other = 0;
public:
    void accept(const InputTuple& tuple) { ①
        if (tuple.word == "lifestyle") lifestyle++;
        else other++;
    }
    bool process() { ②
        vector<OutputTuple> output = {
            {"lifestyle", lifestyle}, {"other", other}
        };
        for (auto& tuple : output)
            emit(tuple); ③
        return true;
    }
};
$$, CountLifestyle;
select * from count_lifestyle( ⑥
    table (select category as word from blog_posts)
);

```

Listing 1: SQL syntax to define and use UDOs: create function defines a function of the language UDO-C++. The C++ code is directly included in the statement, and the function can be called like any other table function by using the table keyword for table arguments.

Figure 3 shows the conceptual overview of the functions that an UDO can implement and use. As the query execution progresses, the user-written functions `accept` ① and `process` ② are executed. There are no additional limitations with the implementation of these functions. As such, the user can create arbitrary control flow by using conditionals and loops. The functions can also call the function `emit` ③ as provided by the UDO User Compiler.

These functions are used to integrate the UDO into the tuple stream of the query. To interoperate with other algebraic operators of the query, the user-written code must:

- (1) Obtain tuples from its input(s), which are other relational operators,
- (2) Process the tuples, i.e., do the actual work, and
- (3) Generate the tuples as output so that the parent operator of the UDO can continue processing the query.

These requirements map nicely to the functions ①, ②, and ③ from above:

Accept, which takes a single tuple from an input and is called repeatedly for each tuple of the input and implements per-tuple processing. *Accept* can, for example, materialize the tuple by storing it in a temporary data structure, or directly compute a (partial) result as shown in ① in Listing 1.

Process, which takes no arguments and is executed after all tuples from the input were seen. It can be used to post-process the input, such as aggregating or sorting it. As the user-written functions can contain arbitrary code and thus arbitrary control flow, more

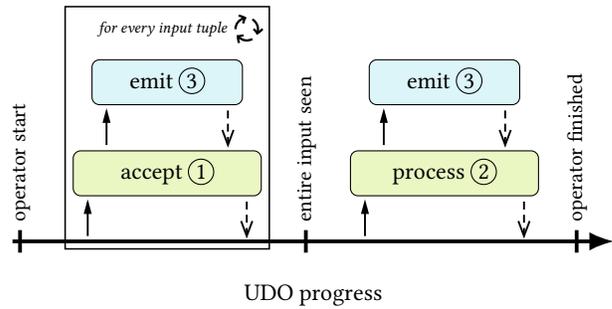


Figure 3: The UDO User Compiler: A user can write code in the `accept` and `process` functions and call the provided `emit` function. Conceptually, `accept` is called once for each input tuple, and `process` is called after the entire input was seen.

complex data analytics algorithms that require the entire input to be available upfront can be implemented as well.

Iterative algorithms, for example, can be easily implemented by using the `accept` and `process` functions. The user code in the `accept` function should store all tuples of the input. The implementation of the `process` function should then use a loop to iteratively compute the result by repeatedly accessing the stored tuples.

Emit, which takes a single tuple as a parameter. This function is provided by the UDO User Compiler and can be called by the `accept` and `process` functions to generate a tuple of the output of an UDO. Conceptually, this tuple is then passed to the parent operator of the UDO.

In the example shown in Listing 1, only `process` ② calls `emit` ③ because the tuples for the output can only be computed once the entire input was seen. The example also shows that inputs and outputs do not have to be equal in their schema or their cardinality. The code only uses a string attribute of the input with an unknown number of tuples while it generates exactly two tuples with a string and an integer attribute as its output.

3.2.1 Usability and Debugging. As the execution of UDFs is usually interleaved with the rest of the query and runs in the same process as the database system, it is not obvious how to enable debugging of the user code. This either requires the database system to implement a debugger itself, e.g., [5], or the user requires access to the database process to use common debugging tools. Both approaches are not optimal. With the former approach users are forced to use DBMS-specific debugging tools instead of their own, and with the latter approach users need low-level privileged access to the database process.

For UDOs there is a more elegant solution: The `accept` and `process` functions are implemented by the user as separate functions in the programming language of their choice. Thus, a test program can be written in that language that provides a tuple stream and calls those functions. Additionally, the test program is required to implement the `emit` function, which is usually provided by the UDO User Compiler. For debugging purposes, the provided function could print its argument, for example.

Our implementation provides a standalone program for C++. It is completely independent from the database system and therefore

```

1 fn  $\Gamma$ .produce():
2   genCode("ht := initialize ht")
3   child.produce()
4   genCode("for r, aggr in ht:")
5   parent.consume("r@aggr")
6 fn  $\Gamma$ .consume(t):
7   genCode("ht.update("+t+")")
8 fn  $\sigma_p$ .produce():
9   child.produce()
10 fn  $\sigma_p$ .consume(t):
11   genCode("if p("+t+"):")
12   parent.consume(t)
13 fn R.produce():
14   genCode("for r in R:")
15   parent.consume("r")

```

Listing 2: Implementation of the produce and consume functions to demonstrate an aggregation (Γ), a selection (σ_p), and a table scan (R) in the produce-consume model.

can be used for any UDO independent of the system the UDO will eventually be used in. To match the characteristics of the execution in a database system more closely, our standalone programs also run multi-threaded and concurrently call the accept and process functions.

3.3 The UDO Query Compiler

The UDO Query Compiler is the part of the system that takes the artifacts generated by the User Compiler, such as a shared library, an object file, or a program of some low-level intermediate language, and integrates it into existing query plans. Additionally, the UDO Query Compiler must implement a standard algebraic operator. This operator must behave just like any other operator in the database system, such as selection, aggregation, or join, so that it can be used in conjunction with other operators in arbitrary queries. As such, the operator must be directly implemented by the query engine of the database system and by design requires the use of database internals.

There are three points where the UDO interfaces with the query engine, which are the three functions accept, process, and emit. The first two functions are implemented by the user, but the emit function must be provided by the UDO Query Compiler. Conceptually, this function takes a tuple that was generated by the UDO and passes it on to the parent operator of the UDO. Depending on the actual execution engine used by the database system, different strategies may be used to implement this function so that it efficiently interacts with the existing engine.

Modern code-generating database systems based on the produce-consume model [22], for example, do not need to implement this function at all. They could replace all functions calls to emit by the actual code that is generated in the parent operator. We provide a detailed description of an implementation of UDOs in a code-generating query engine in Section 4.

Database systems that employ the iterator model [11, 19] such as Postgres can implement the emit function by storing the emitted tuples in a temporary buffer. When the execution of the UDO is finished, the buffer can be used to return the tuples to the parent operator of the UDO. The same strategy can be used in vectorized query engines that are used in systems like MonetDB [6], Vectorwise [33], or DuckDB [26].

Materializing all tuples in a temporary buffer in the iterator model may add significant runtime overhead. This overhead can be

```

 $\Gamma$ 
 $\sigma_p$ 
R
 $\Gamma$ .produce():
 $\sigma_p$ .consume(r):
R
 $\Gamma$ .produce():
R
 $\Gamma$ .produce():
 $\sigma_p$ .consume(r):
 $\Gamma$ .consume(r):
 $\Gamma$ .produce():

```

```

1 ht := initialize ht
2 for r in R:
3   if p(r):
4     ht.update(r)
5   for r, aggr in ht:
6     output r@aggr

```

Figure 4: Code generated by a query compiler in the produce-consume model. The code generated by the algebraic operators is interleaved which leads to better data locality.

avoided by interrupting the user code as soon as it calls the emit function. Then, the single generated tuple can be passed to the parent operator. When the UDO should generate more tuples, the interrupted code must be continued. We explain how this approach can be implemented in more detail in Section 5.

4 USER-DEFINED OPERATORS IN CODE-GENERATING QUERY ENGINES

In this section we present our implementation of UDOs in our database system Umbra [23]. Umbra is a main-memory first system, which is geared towards working primarily in memory but also supports storing relations on disks or preferably fast SSDs. After translating SQL to relational algebra, Umbra uses the produce-consume model [22] to generate efficient code for the query. As code generation can be relatively expensive, especially for ad-hoc queries that complete quickly, Umbra does not directly generate machine code but uses the intermediate representation *Umbra IR*. This low-level language that was inspired by LLVM [17] can then be executed in multiple ways: A virtual machine that interprets the IR, a direct translation from Umbra IR to x86 assembly called the *Flying Start* backend [15], and a more sophisticated translation that uses LLVM to generate optimized machine code. Umbra uses *adaptive execution* [16] to dynamically switch between all three approaches to achieve low latency for short queries and high throughput for long-running analytical queries. Finally, to enable good scalability across many CPU cores and even NUMA nodes, our execution engine employs *morsel-driven parallelism* [18].

4.1 UDO User Compiler

The goal of our implementation is for queries containing UDOs to run as fast as “native” queries that consist of only Umbra IR. To achieve this goal, our implementation supports C++ as the programming language for user code. As a systems language, it can be directly compiled to efficient native machine code. The second reason why we chose C++ specifically is that it can also be compiled to LLVM IR with the Clang compiler. As our compilation framework can use LLVM IR as well, this makes it possible to completely inline user code into the generated machine code, thus enabling native query speed.

Our UDO User Compiler for C++ provides some definitions of classes and functions that can be used. The conceptual accept and process functions are realized as member functions of a base class which the user must derive. The emit function is another predefined member function of this class. As the functions that the user writes are member functions of a class, the UDO code can also make use of

```

1 fn UDOa.produce():           7 fn UDOa.consume(t):
2   child.produce()           8   genCode("accept("+t+")")
3   if UDO has process:
4     genCode("process()")
5     genCode("fn emit_helper(t):")
6     parent.consume("t")

```

Listing 3: Implementation of UDO_a in the produce-consume model.

member variables to manage state across invocations of the accept and process functions. To enable parallelism within the UDO, our system potentially calls accept and process concurrently. Therefore, users must ensure that those functions are thread-safe.

The example in Listing 1 shows C++ code that our UDO User Compiler can use. The user-written class CountLifestyle is a subclass of UDOOperator, which uses two member variables that track the number of occurrences of the word “lifestyle” and all other words. The variables are updated in accept ① and used for the generated output in process ②. The emit function is called with a single output tuple as an argument ③.

For the UDO Query Compiler our User Compiler generates two artifacts generated from the C++ code: An object file and an LLVM module. The object file uses the ELF format and could theoretically also be generated by any other C++ compiler that supports ELF. The LLVM module is specific to the Clang compiler. The module is generated, so that the UDO Query Compiler can potentially inline the UDO code with the rest of the query code generated by the database system.

Even completely different programming languages that do not use LLVM at all could be used. Our UDO Query Compiler in Umbra can work with any code that can be compiled to ELF object files. Some optimizations are not possible when the LLVM module is not available, nevertheless the UDO can be executed.

4.2 UDO Query Compiler

Our UDO Query Compiler takes the object file and the LLVM module from the UDO User Compiler and integrates it into a query plan. In the query, the UDO is represented as a relational algebra operator, which we will call UDO_a. The implementation of UDO_a is not specific to one particular UDO. UDO_a can take any UDO that was processed by the UDO User Compiler and integrate it into existing queries.

Because it is treated like any other existing algebraic operator, UDO_a integrates seamlessly with all DBMS components, such as the optimizer. While general optimizations across UDO_a are not possible since the optimizer does not have any information about its semantics, the subtree that represents the input of UDO_a and the subtree that contains UDO_a can still be optimized.

4.2.1 Produce-Consume Model. Umbra uses the produce-consume model [22] to generate code for a relational algebra tree. In this model, every algebraic operator implements the two functions produce and consume that generate code. Listing 2 shows how these functions could be implemented for an aggregation that uses hash tables (Γ), a selection with a predicate p (σ_p) and a table scan for

the relation R . In general, the produce function is called when an operator should start generating its output. For Γ , this function first generates code to initialize a hash table, and then calls the produce function on its input. When an operator wants to pass a tuple of its output to its parent, it calls the consume function of the parent. The Γ operator generates code to update the hash table with the new tuple in its consume function. The remainder of Γ .produce() then iterates over the hash table and calls the consume function of its parent with the aggregated result. The implementation of the σ_p operator is simpler; the selection does not need to initialize any data structures, so it only calls the produce function of its input in σ_p .produce(). σ_p .consume() generates code to evaluate the predicate and calls the consume function of its parent that will generate code in the true-branch of the predicate.

Figure 4 shows a query that uses those three operators and the generated code. Code from different operators tends to be interleaved. This approach leads to good data locality and very efficient execution on modern hardware but makes it difficult to integrate “foreign” code.

4.2.2 Generating Code for UDOs. To integrate the functions ① – ③ of an UDO into a query plan, the UDO Query Compiler must generate code that acts as an interface between the user code and the code generated by built-in operators. As we model UDO_a like any other algebraic operator, there are two choices to emit code: the produce function and the consume function.

Integrating accept is straightforward; the consume function must generate code to call accept. Conceptually, the consume function generates code that processes a single tuple and we defined accept as a function that takes a single tuple, so they are a perfect match. Similarly, the process function must be called in the code generated by produce. Only then, UDO_a can make sure that the entire input was seen and accept was called with all tuples from the input before the process function is called for the first time.

Listing 3 shows the implementation of UDO_a in the produce-consume model. In addition to calling accept and process in consume and produce, respectively, UDO_a.produce() also generates code that defines a new, separate function emit_helper(). This function is used by the UDO Query Compiler to implement emit.

Every call to emit in the user code means that a new tuple of its output is generated. This tuple must eventually reach the parent of UDO_a so that the query processing can continue. In the produce-consume model, the parent receives an input tuple in the consume function. To bridge the gap between the user code and generated code, UDO_a emits the code of its parent into the separate function emit_helper(). Then, the UDO Query Compiler replaces all calls to emit in the user code by calls to emit_helper().

With this approach, the UDO is entirely transparent both to the user code and the other relational algebra operators; they do not require any implementation details about the other. Built-in operators can call produce and consume of UDO_a as usual, and the user code uses the API functions ① – ③.

Figure 5 shows the algebra tree and the generated code for a query that contains an UDO. This query is similar to the query from Figure 4; only the selection was replaced by an UDO. For this example, we assume that the UDO only calls the emit function in accept and does not implement process. While the generated

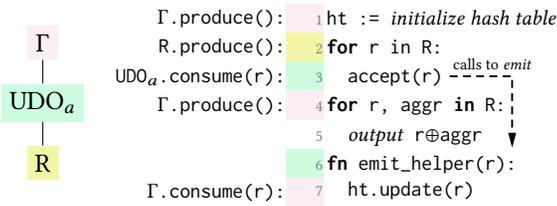


Figure 5: Code generated by the UDO Query Compiler and UDO_a for an UDO that behaves like a selection (i.e., “pipelined operator”). Calls to emit in the user code are replaced by calls to the generated emit_helper function.

code defines the function emit_helper(), it does not contain calls to emit_helper(), because emit_helper() is only indirectly called every time the UDO code calls emit.

Figure 6 shows another algebra tree and its generated code. Again, the query from Figure 4 is shown but we replaced the aggregation with an UDO. Additionally, the UDO now calls emit only in the process function. Writing UDOs that call emit only in process is useful for algorithms that can only generate their output once the entire input is seen. The generated code contains a call to process after the loop of the table scan. Thus, the process function will only be called after the UDO received all tuples of its input.

4.3 UDO Function Inlining

The examples in Figures 5 and 6 show that, initially, UDO_a introduces several low-level function calls that will be executed for each tuple of the input and output; The accept function is called in line 3 for every tuple and the user code calls emit and indirectly emit_helper for every tuple of its output. When UDOs are executed in main-memory systems running on modern hardware, the best performance can only be achieved when the number of function calls is reduced. As data can be processed so quickly, every function call adds noticeable overhead. To solve this issue, we use a very common approach used in compiler optimization; We inline the function calls to eliminate them.

This approach is possible in our implementation because we can compile the user code written in C++ to LLVM. Umbra supports different execution modes, which includes LLVM, as well. Hence, we generally compile the user code to native machine code, which is stored as object files. To prepare the UDO code to be eventually inlined, the C++ code is compiled to LLVM.

When the query is executed, the object file is loaded into memory and the code generated by UDO_a uses real low-level function calls into the functions that are located in the object file. Since the object file is generated only once when the user runs create function, this compilation incurs no compilation overhead from the high-level compiler, which for languages like C++ could take up to several seconds. In fact, since the object file is generated only once for every UDO, we can afford running all available compiler optimizations on the user code. This means that even though the compiled query code will execute lots of function calls to accept, process, and emit, the UDO code is very efficient.

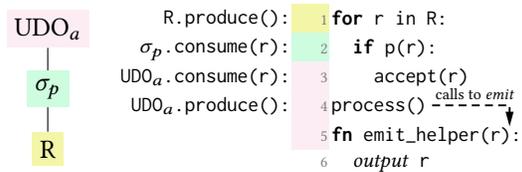


Figure 6: Code generated by the UDO Query Compiler and UDO_a for an UDO that behaves like an aggregation (i.e., “pipeline breaker”). The generated code also calls accept and additionally makes sure that process is called only after the entire input is seen.

When our framework for adaptive execution [16] detects a long running query, our compilation engine generates LLVM code for the entire query plan. When that happens, we search for all call instructions that call the accept, process, or emit functions from the object file and replace them by their LLVM representation.

Note that in Umbra, switching to the LLVM mode is not a static decision that must be made before executing the query. This decision is made by the execution engine at runtime. The execution of the UDO code is transitioned smoothly with no downtime from the unoptimized implementation, which uses real functions calls into the object files, to the optimized and inlined LLVM code.

Listing 4 shows a part of the LLVM code, which is generated by our UDO Query Compiler. This code was generated from the SQL query shown in Listing 1 that selects from a relation and uses an UDO. The part shown here contains the table scan and the code for accept. The resulting LLVM code has no clear boundaries between the user code and the code generated for the table scan and thus has good locality. The first part of the code loads the string value from the column of the relation. This code is directly generated by the query engine for the table scan. It is followed by the inlined UDO code which first checks if the string has exactly 9 characters, because it compares it with the word “lifestyle”. If it does, it then actually compares the string by using the memcmp function and increments the corresponding counter. The rest of the code, again generated for the table scan, increases the tuple index, and repeats the loop if any tuples are left.

Independent from the actual implementation, the concept of function inlining can quickly lead to an increase in code size especially when inlined functions contain calls to other functions that are inlined. In the UDO Query Compiler, this issue can occur when a single query contains multiple UDOs. To avoid any code explosion caused by inlining, we strictly allow the UDO code to call emit only exactly once syntactically. As a result, there is no potential for code to be duplicated; therefore, code explosion can be avoided. This does not mean that it is impossible to call that function multiple times semantically. Wrapping the call to emit in a loop, for example, still satisfies our requirement but at runtime the function is called multiple times. In general, this restriction could be lifted automatically by a compiler. Multiple calls to emit could be replaced by a goto statement that jumps to one common code location where, again, emit is called only once.

```

scan_loop_head:
    %tuple_index = phi i64 [ i64 0, %start_scan ],
    [ %next_index, %string_eq_block ]
    %attr_ptr = getelementptr { i64, i64 },
    { i64, i64 }* %column_ptr, i64 %tuple_index,
    i32 0
    %str_header = load i64, i64* %attr_ptr, align 8
    %str_body = getelementptr inbounds i64,
    i64* %attr_ptr, i64 1
    %str_offset = load i64, i64* %str_body, align 8
    %str_len = trunc i64 %str_header to i32
    %is_long_str = icmp ugt i32 %str_len, 12
    %str_raw_ptr = select i1 %is_long_str,
    i64 %str_data, i64 0
    %str_ptr = add i64 %str_raw_ptr, %str_offset
    store i64 %str_header,
    i64* %local_var_str_header, align 8
    store i64 %str_ptr,
    i64* %local_var_str_body, align 8
    %has_len_9 = icmp eq i32 %str_len, 9
    br i1 %has_len_9, label %cmp_str_block,
    label %else_block
cmp_str_block:
    %string_cmp = call @memcmp(i8* %local_var_str,
    i8* @str_literal, i64 9)
    %string_eq = icmp eq i32 %string_cmp, 0
    br i1 %string_eq, label %string_eq_block,
    label %string_ne_block
string_ne_block:
    br label %string_eq_block
string_eq_block:
    %var = phi i8* [ %dbs_var, %string_ne_block ],
    [ %nonddb_var, %cmp_str_block ]
    %counter = bitcast i8* %var to i64*
    @atomicrmw add i64* %counter, i64 1 monotonic
    %next_index = add i64 %tuple_index, 1
    %at_end = icmp eq i64 %next_index, %relation_size
    br i1 %at_end, label %finish_scan,
    label %scan_loop_head

```

Listing 4: LLVM code generated by the UDO Query Compiler after inlining the UDO from Listing 1. The user code is directly interleaved with the rest of the query code which removes all potential function call overhead.

4.4 Parallel Execution

Generally, Umbra generates code that processes all queries concurrently on all available CPU cores. To achieve the best performance, UDO_a must also generate code that can be executed concurrently.

Our code generation framework and the morsel-driven scheduler ensure that code generated by the consume function of any relational algebra operator is called concurrently. As our implementation of consume for UDO_a generates a call to accept, this means that the user-written code in accept is executed concurrently as well. Similarly, we generate code that calls the process function concurrently on all available CPU cores.

For this approach to work, the functions accept, and process should be thread-safe. Therefore, the user must ensure that those functions can be called in parallel. Only thread-safe data structures

or common idioms for synchronization such as mutexes or atomic operations should be used. Furthermore, since these functions may contain calls to emit, our UDO Query Compiler ensures that emit is thread-safe, as well. When functions are implemented as UDOs to compete with code-generating operators directly implemented in Umbra, the user code must be written very well and use state-of-the-art synchronization.

4.5 Implementation Considerations

In this section we discuss some additional technical details and considerations for our implementation. While they do not extend the theoretical framework of the UDO User Compiler and UDO Query Compiler, they are still useful to obtain a full picture of our implementation.

4.5.1 Global State and Concurrent Queries. As user code can be arbitrary code, it can also make use of global variables. While it is generally considered bad practice to rely heavily on global state, especially when global state is mutated, it is sometimes useful or even necessary to use. The standard library of C++, for example, uses global state to implement some functions more efficiently. Another use of global state are *thread-local* variables, which are often used to make parallel implementations of algorithms more efficient.

As we allow users to write arbitrary code, our UDO User Compiler does not prohibit the use of (thread-local) global variables. Because we also want to maintain the isolation of separate queries running concurrently to not violate ACID properties, our UDO Query Compiler maintains entirely separate global and thread-local states for every instance of an UDO that is being executed. Thus, when a new query that contains an UDO starts, its global state will always reflect a clean state and is not affected by any other queries that use the same UDO.

4.5.2 Linking of Runtime Dependencies. Executing C++ and even C code requires several dependencies to be loaded at runtime. Most notably the C standard library, also called *libc*, and the C++ standard library must be loaded so that the user code can use all features provided by the language. The trivial approach of loading them as shared libraries into the database process is not feasible as this would interfere with the rest of the system. Additionally, this approach does not allow separating the global states of those libraries. Just like global variables explicitly written by the user, we also want to provide full isolation of all runtime dependencies so that two UDOs running concurrently can never interfere with each other.

To solve this issue, our system contains a custom runtime linker, which can load the required runtime libraries as object files or static libraries, and load them into the existing database process. This linker also takes the object file that is generated from the UDO code and links it with the runtime dependencies. Finally, the linker also supports allocating global and thread-local state which allows us to implement the strict separation for global states of concurrent UDOs.

4.5.3 Security. The code generated by the built-in relational operators is written by experts and is carefully tested. This means that bugs tend to be rare so that arbitrary SQL queries can be safely executed. UDOs, however, are potentially written by users

less familiar with the system or even malicious users. Especially in Umbra, where the generated code is inlined directly into the remaining query code, it has the same capabilities and privileges. Therefore, bugs in an UDO can crash the entire database system, and malicious actors can use this as an easy privilege escalation. To prevent memory bugs, the Rust language [2] can be used. The Rust compiler uses LLVM, and can guarantee memory-safety. Solving the issue with malicious code is not trivial, especially if the same performance characteristics should be maintained. For now, UDOs in our implementation must not be used with untrusted code.

5 USER-DEFINED OPERATORS IN THE ITERATOR MODEL

The concept of UDOs is not limited to being implemented in code-generating databases. Query engines based on the iterator model can achieve very efficient execution of UDOs, as well. This approach allows the efficient execution of custom algorithms in traditional disk-based database systems. In this section, we present our implementation of UDOs in Postgres.

In the iterator model, every algebraic operator defines a next function. This function generates a single tuple of the output of the operator. It is called repeatedly until the entire output is generated, which establishes a *pull-based* control flow, that is, the parent operator decides when the child operator should generate the next tuple. This stands in contrast to the architecture of UDOs; the user code can decide by itself when to generate a new tuple of the output and call emit. Thus, UDOs have a *push-based* control flow; children operators push their outputs to their parents. An implementation of UDOs in a query engine based on the iterator model must bridge the gap between the push-based user code and the pull-based execution of the query in which the UDO is contained.

5.1 UDO User Compiler

As the UDO User Compiler can generally be independent of the actual query engine of the existing database, it does not have to be reimplemented for every database. Our Postgres implementation uses the exact same UDO User Compiler as our implementation in Umbra (refer to Section 4.1). As such, it supports user-written C++ code. The UDO User Compiler in Umbra can also make use of the LLVM IR to further optimize the code generated for queries that contain UDOs. Postgres does not generate code for entire queries, so it only uses the object files generated by the UDO User Compiler.

5.2 UDO Query Compiler

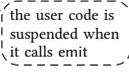
As mentioned above, the main problem that the UDO Query Compiler, which is integrated into the iterator model, must solve is the mismatch between the pull-based query engine and the push-based user code. Conceptually, the implementation of the algebraic operator UDO_a must first call `accept` with all tuples of the input and then call `process`. In the context of the iterator model, this is implemented in the next function of UDO_a .

Listing 5 shows how the functions of the UDO are called in the iterator model. The next function must call the `accept` function for every tuple of its child operator. These tuples are fetched by calling the next function of the child in the loop in lines 9 and 10. After all tuples are fetched, the `process` function is called in line 11.

```

1 fn UDOa.next():
2   if UDO not finished:
3     if state is empty:
4       state = init next_coro()
5       resume state
6     if state is suspended:
7       return state.tuple
8   coro UDOa.next_coro():
9     while child has tuples:
10      accept(child.next())
11      process()

```



Listing 5: Implementation of UDO_a in the iterator model.

The iterator model mandates that the next function should return a single tuple. To avoid memory and runtime overhead, $UDO_a.next()$ should not store any intermediate results in temporary buffers. However, the user code can decide arbitrarily when to call `emit` either in `accept` or `process`. Hence, the implementation of $UDO_a.next()$ does not have control over when a new tuple is generated. To solve this issue, our implementation *suspends* the execution of the user code as soon as it calls `emit`. It saves the execution state, such as the stack and the instruction pointer, of the user code and jumps back to where the execution of the user code was first started in line 5. It also remembers the tuple argument that the user code passed to `emit` so that it can return the tuple in line 7.

Conceptually, we treat the `next_coro` function (lines 8 to 11) as a *coroutine*. Instead of calling it once and getting a result value once, we first *initialize* it in line 4. This sets up the execution state for the coroutine but does not yet execute the function. In line 5, the coroutine is *resumed*, which means that the execution starts (for a new coroutine) or continues (for an old coroutine that was suspended before). The execution continues until the coroutine is *suspended*. In our implementation, this happens precisely when the user code calls `emit`.

Note that the coroutine is an implementation detail of the UDO Query Compiler in the iterator model. The user code is not modified in any way to enable the execution of the coroutine. As our implementation in Postgres uses the same UDO User Compiler as in Umbra, the exact same user code can be used in both systems.

6 EVALUATION

To evaluate our implementation, we implemented several different functions as UDOs and executed them in Umbra and Postgres. We compared their runtime with equivalent queries implemented in standard SQL or “native” operators of Umbra that generate code. We also ran some queries on Spark, a data analytics engine, and DuckDB, an in-memory database system that uses vectorized execution. Our results show that the runtime of queries containing UDOs is always similar to or faster than all competing approaches while allowing the user to write standard C++ code.

We ran all our benchmarks on a NUMA machine with two Intel® Xeon® E5-2680 CPUs with 14 cores and 28 hyper-threads each and 128 GiB of DRAM per node. Unless otherwise noted, we ran all benchmarks on all 56 hyper-threads.

We ran all queries 10 times and reported the median. We ensured that the data sets were loaded into the file system caches before running the queries. Postgres is configured to use 128 GiB of DRAM. Spark is configured to use 64 GiB of main memory each in the driver and executor.

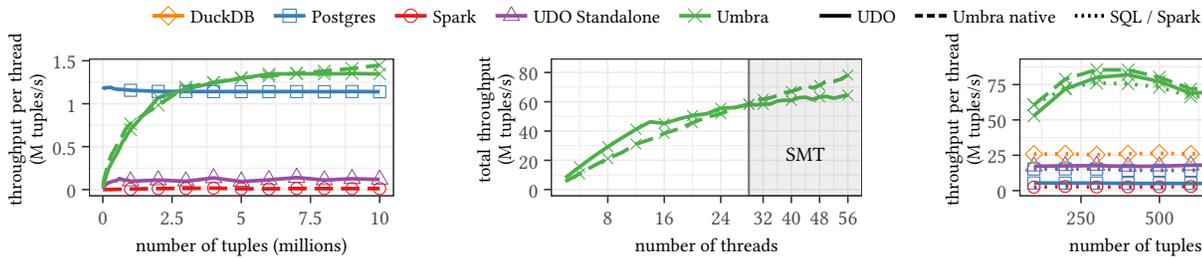


Figure 7: Throughput per thread of k-Means: Postgres runs single-threaded but can reach a reasonable throughput on that thread by using a UDO.

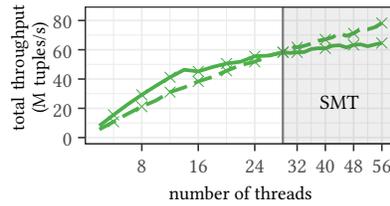


Figure 8: Total throughput of k-Means with varying number of threads: The UDO can scale just as well as the native code-generating implementation in Umbra.

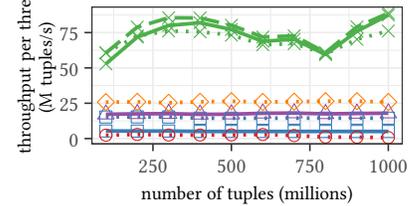


Figure 9: Throughput per thread for different implementations of simple linear regression. For aggregation-like algorithms, code-generation surpasses all other approaches.

6.1 Complex Iterative Algorithm: k-Means

One of the main use cases of UDOs is to integrate complex data analytics algorithms that are written in imperative code into the database system. We want to show that implementing such an algorithm as an UDO can achieve the same performance as if it were implemented directly into the database by generating code. For nonexperts of our database system, it is not feasible to implement a new operator on the relational algebra level, especially for complex algorithms. To be able to understand the performance characteristics of UDOs, however, it is best to have a direct comparison of UDO vs. native code-generating code.

For this we chose the comparatively simple k-Means algorithm and implemented it both as a C++ UDO and as a native operator directly into Umbra. Conceptually, both implementations follow this simplified algorithm:

- (1) Initialize cluster centers by randomly sampling k points.
- (2) For each point select the cluster center with the smallest distance and update the point's cluster id.
- (3) For each cluster find all points with the same cluster id and calculate the new cluster center.
- (4) Repeat steps (2) and (3) 10 times.

Usually, the iteration is stopped when a cancellation criterion is met, such as a required minimum movement of the cluster centers. As this depends heavily on the initialization of cluster centers, which is random in our implementation, instead we always iterate exactly 10 times to ensure that the runtimes of all approaches are directly comparable.

We use synthetically generated two-dimensional points that are clustered in eight clusters as the data-set for our benchmarks. The points within each cluster are drawn from a normal distribution where each cluster has separate means and variances. As the number of iterations is fixed to ten, all executions always must scan all points exactly ten times and compare them to the current cluster centers of all eight clusters.

The implementation effort required for both approaches – one natively in Umbra, using code-generating code and database internals, and the other as a self-contained UDO – highlights the qualitative advantage of using UDOs. The Umbra implementation extends the relational algebra by adding a new algebraic operator

that implements k-Means. Adding a new operator requires modifying several parts of the system such as the SQL parser, the optimizer, and of course the query engine. In its core, the implementation consists of about 500 lines of code of which many generate one or more instructions. The UDO implementation has 300 lines of self-contained C++ code. This code only uses features from the C++ standard library and some auxiliary data structures. As the UDO does not use any database-specific code, it can easily be compiled into a standalone executable program which also makes debugging much easier.

Spark allows for even more straightforward implementation of data analytics algorithms such as k-Means as they are directly built-in into the system. At its core, the Spark code consists of only one call to the existing k-Means clustering function and a few more lines to set up the experiment. When compared to writing C++ code for UDOs, Spark enables users without in-depth knowledge of programming languages such as C++ or Rust to do data analytics. However, Spark cannot reach the performance of queries using UDOs.

The main advantage of using UDOs in existing database systems instead of specialized data analytics systems like Spark is their execution speed. Figure 7 shows the throughput per thread of different implementations of the k-Means algorithms as described above for data sets containing up to 10 million tuples. We ran the C++ UDO in our UDO implementations for Umbra (multi-threaded) and Postgres (single-threaded). Additionally, we tested the performance of the standalone executable that is mainly used for debugging (refer to Section 3.2.1).

As a comparison, we provide measurements of a native k-Means operator of Umbra, which generates efficient code and makes heavy use of database internals to achieve good parallelization. Interestingly, the UDO executed in Umbra almost reaches the performance of the native operator. Even though the k-Means UDO uses standard C++ constructs, the UDO User Compiler and the UDO Query Compiler can generate code for it and optimize it to match the specialized code-generated implementation.

To demonstrate the scalability of the UDO implementation onto many threads, we compare the runtime of the k-Means query that processes 500 million tuples while varying the number of threads. Figure 8 shows the total throughput for this query for the native and the UDO implementations in Umbra. As long as not more than

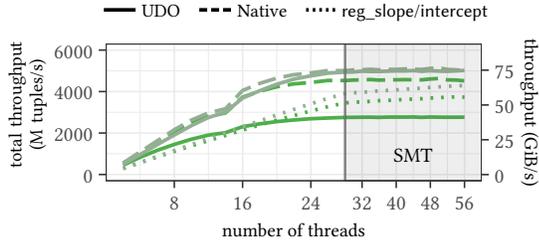


Figure 10: Total throughput of simple linear regression in Umbra with varying number of threads. The faded lines show the raw throughput without compilation overhead. The UDO query reaches over 70 GiB/s but its compilation overhead brings down the total throughput.

half the threads are used so that no SMT must be used, the UDO can outperform the native operator.

The reason why the UDO implementation can outperform custom-written code-generating code is that it uses a standard high-level language compiler. The C++ code is compiled by using the Clang compiler, thereby enabling all available optimizations. Umbra generates the low-level code directly without going through a higher-level language first and uses fewer expensive optimizations than the Clang compiler. For UDOs, spending more time on compiling C++ is not a performance issue as this is only done once when the create function statement is executed. A code-generating database must balance the trade-offs between spending more time to generate faster code.

The figure also includes the throughput of an equivalent k-Means query executed in Spark. Our implementation warms up Spark by loading the input data into main-memory and executing the clustering once without measurement. As for the other implementations, it randomly selects points to initialize the cluster centers and iterates 10 times.

The throughput of both Umbra and the UDO implementations is almost 50 times higher than in Spark. There are several reasons for this increased throughput. Spark runs in the JVM which does support JIT compilation to native machine code but must compete with ahead-of-time compiled, optimized C++ code. Furthermore, Spark is designed to work on multi-node clusters which means that its single-node performance may not be optimal. Its parallelization and synchronization model is also designed to make it easy to scale out to many servers but cannot easily benefit from intraprocess synchronization.

6.2 Linear Regression

To benchmark another algorithm used in data analytics, we tested simple linear regression using least squares. We implemented it as an UDO, natively in Umbra, Spark, and SQL. SQL offers the functions `regr_slope` and `regr_intercept`, which can be used for simple linear regression on a linear function. All Non-SQL implementations use a polynomial of degree 2 as a target function to highlight the use case for hyperparameter tuning, which often requires slight changes to existing algorithms.

Our implementations solve the following problem: For given pairs of values x, y choose a, b , and c while minimizing the error

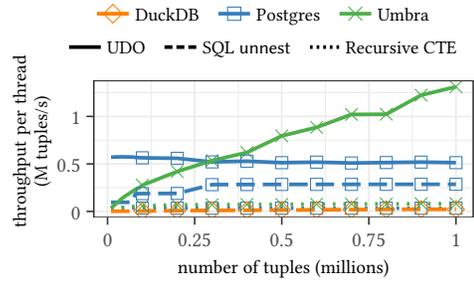


Figure 11: Throughput of a query that splits comma-separated values into individual tuples. When the inherent imperative control-flow is simulated by using recursive CTEs in SQL, efficient code cannot be generated.

term $\sum_i (a + bx_i + cx_i^2 - y_i)^2$. This problem has the following closed-form solution:

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} \sum 1 & \sum x & \sum x^2 \\ \sum x & \sum x^2 & \sum x^3 \\ \sum x^2 & \sum x^3 & \sum x^4 \end{pmatrix}^{-1} \cdot \begin{pmatrix} \sum y \\ \sum xy \\ \sum x^2y \end{pmatrix}$$

To compute this efficiently, first, all sums need to be calculated. Calculating the sums requires little synchronization and should scale very well. The values for a, b , and c can be determined at the end by calculating the inverse of the matrix.

Figure 9 shows the throughput per thread for all implementations. As this algorithm is essentially an aggregation, the very fast code-generating query engine used in Umbra surpasses all other approaches. This result, of course, is not due to the use of UDOs but because generating code is the most efficient approach to compute this algorithm. Still, the throughput of the UDO implementation in Umbra is very similar to the native, code-generating operator.

We tested the different implementations in Umbra in more detail, as can be seen in Figure 10. We ran the query on the data set with 10^9 tuples (16 GB) and varied the number of threads. The plot shows that the maximal throughput is reached well before all threads are used, because Umbra can saturate the memory bandwidth with all three approaches when using only half of the threads.

Figure 10 also shows one negative effect of using UDOs: When the total throughput (green lines) is compared to the throughput excluding compilation overhead (faded lines), the query using the UDO performs the worst. This result is because the compilation time for a query containing an UDO (≈ 180 ms) is significantly higher than for queries without UDOs (≈ 37 ms). When an UDO is inlined, the resulting query plan generally contains more LLVM instructions as they are generated by a high-level C++ compiler. Umbra's native operators directly generate more concise code, which does not require many optimizations.

6.3 Imperative Programming

To highlight the advantages of using an imperative language to process queries, we tested a query that generates multiple output tuples for every input. The input contains a string that is a comma-separated list of numbers and words. The UDO parses this string, takes out all the numbers, and generates a new tuple for each number.

```

with recursive split_arrays(name, value, tail) as (
  select c.name, NULL, c.values as tail
  -- schema: array_values(name text, values text)
  from array_values c
  union all
  select s.name,
         case when comma = 0 then s.tail
         else left(s.tail, comma - 1)
         end as value,
         case when comma = 0 then ''
         else right(s.tail, -comma)
         end as tail
  from (
    select s.*, position(',', ' in s.tail) as comma
    from split_arrays s
  ) s where s.tail != ''
)
select name,
       case when value similar to '[0-9]+'
       then cast(value as bigint)
       else null end as value

```

Listing 6: Splitting comma-separated strings into individual integer tuples using recursive CTEs in SQL.

While the C++ code for this algorithm is only a few lines, implementing this query in SQL is very tedious. Listing 6 shows how recursive CTEs can be used to formulate this conceptually simple query. Every input tuple can generate an arbitrary number of outputs; however, since SQL does not support loop statements, recursion must be used instead. To ensure that the cast is not evaluated for invalid strings, a case when statement must be used. Moving the similar to expression to the where clause would allow the database to reorder the expressions which could lead to runtime errors when the cast is evaluated before the similar to expression.

In DuckDB, this recursion can be prevented by using the special function `unnest()`, which directly converts an array of values into multiple tuples. Postgres has a similar function called `string_to_table()`, which splits a string by a given separator into multiple tuples.

For such an algorithm that is inherently imperative, the UDO can easily outperform the SQL versions for larger data sizes. Figure 11 shows the throughput for ad-hoc queries, so it includes the compilation time. When the control flow from the imperative C++ program must be simulated by using a recursive CTE, even Umbra is not able to generate efficient tight loops. This leads to slow executions that cannot compete even for small input sizes where the runtime of the UDO is dominated by its compilation time.

6.4 Data Generation

Our implementation does not require UDOs to have any inputs. Therefore, it is possible to write “output-only” functions that can take scalar arguments and then return a stream of tuples, that behave similar to functions like `generate_series()` in SQL.

When writing benchmarks, it is very common to synthetically generate all data that is used by the test queries. Such data generators are usually written in an imperative language. Examples for this are C for the widely used benchmarks TPC-H and TPC-DS, and Python, which has a broad range of libraries for that purpose.

Table 1: Runtimes to generate and insert different benchmark data sets by using UDOs.

Data Set	Tuples	Size	Insert	Generate
points (double, integer)	10M	354 MiB	5 s	0.4 s
points (double, integer)	100M	3.5 GiB	52 s	2 s
comma-separated (text)	10M	700 MiB	15 s	13 s
comma-separated (text)	100M	7 GiB	148 s	123 s

The disadvantage of this approach is that the data is generated first by a program and then it must be inserted into a database. With UDOs this can now be implemented directly in the database. No extra steps to export and import data are required.

All test data for our benchmarks was created by using UDOs which were directly used in INSERT statements. Table 1 shows the runtime of some insert statements for different data sets. The sizes shown in the table refer to the size the generated data set would have if exported as a CSV file. For the “words” and “comma-separated” data sets, which contain only strings, insert queries using an UDO as a source can process around 40 MiB/s to 50 MiB/s. This result is mainly a limitation caused by the insert statement, not by the UDOs themselves. The “Generate” column shows how long it takes to just generate the tuples and immediately discard them. The two UDOs that make heavy usage of string operations can reach up to 80 MiB/s. The points dataset that does not use any strings reaches about 1.7 GiB/s which means it can keep up with modern SSDs and the runtime of the data generation will most likely be dominated by the actual insertion of tuples into a physical relation.

All cases have in common that they do not require writing any intermediate files to disk that must then be read again by the database system. Especially for larger data sets, this approach prevents wasting space and time to store data on disk that is quickly discarded. This approach yields no disadvantages for the user as this data generation can be written in standard, imperative code.

7 CONCLUSION

In this work, we presented *User-Defined Operators* (UDOs) – a novel framework to efficiently integrate and execute custom algorithms in modern databases. UDOs can achieve very high throughput, which is competitive with main-memory databases. Furthermore, because UDOs are integrated into query engines of existing RDBMS, all ACID properties can be preserved. Nevertheless, users are not required to know any database internals. Instead, they are provided with an easy-to-use API.

We implemented UDOs in Postgres and Umbra – a code-generating database. Our evaluation shows that queries containing UDOs can achieve throughputs similar to main-memory databases. Even in disk-based systems such as Postgres, the execution of UDOs is very efficient. Thus, UDOs enable users to integrate custom algorithms for data analytics directly into databases very efficiently.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 725286). 

REFERENCES

- [1] 2022. PL/pgSQL – SQL Procedural Language. Retrieved January 14, 2022 from <https://www.postgresql.org/docs/14/plpgsql.html>
- [2] 2022. Rust Programming Language. Retrieved January 14, 2022 from <https://www.rust-lang.org/>
- [3] 2022. Transact-SQL Reference. Retrieved January 14, 2022 from <https://docs.microsoft.com/en-us/sql/t-sql/language-reference>
- [4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016).
- [5] Yanif Ahmad and Christoph Koch. 2009. DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases. *Proc. VLDB Endow.* 2, 2 (2009), 1566–1569.
- [6] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. www.cidrdb.org, 225–237.
- [7] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. 2015. An Architecture for Compiling UDF-centric Workflows. *Proc. VLDB Endow.* 8, 12 (2015), 1466–1477.
- [8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
- [9] Christian Duta, Denis Hirn, and Torsten Grust. 2020. Compiling PL/SQL Away. In *CIDR*. www.cidrdb.org.
- [10] Eric Friedman, Peter M. Pawlowski, and John Cieslewicz. 2009. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.* 2, 2 (2009), 1402–1413.
- [11] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*. IEEE Computer Society, 209–218.
- [12] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding their usage in the wild. *Proc. VLDB Endow.* 14, 8 (2021), 1378–1391.
- [13] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711.
- [14] Denis Hirn and Torsten Grust. 2021. One WITH RECURSIVE is Worth Many GOTOs. In *SIGMOD Conference*. ACM, 723–735.
- [15] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905.
- [16] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *ICDE*. IEEE Computer Society, 197–208.
- [17] Chris Latner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE Computer Society, 75–88.
- [18] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.
- [19] Raymond A. Lorie. 1974. XRM - An Extended (N-ary) Relational Memory. *Research Report / G / IBM / Cambridge Scientific Center G320-2096* (1974).
- [20] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: a timely dataflow system. In *SOSP*. ACM, 439–455.
- [21] Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. 2016. Incremental, iterative data processing with timely dataflow. *Commun. ACM* 59, 10 (2016), 75–83.
- [22] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.
- [23] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.
- [24] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (2018), 1002–1015.
- [25] Linnea Passing, Manuel Then, Nina Hubig, Harald Lang, Michael Schreier, Stephan Günemann, Alfons Kemper, and Thomas Neumann. 2017. SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In *EDBT OpenProceedings.org*, 84–95.
- [26] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD Conference*. ACM, 1981–1984.
- [27] 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*. ACM, 6:1–6:12.
- [28] Lujia Yin, Yiming Zhang, Zhaoning Zhang, Yuxing Peng, and Peng Zhao. 2021. ParaX: Boosting Deep Learning for Big Data Analytics on Many-Core CPUs. *Proc. VLDB Endow.* 14, 6 (2021), 864–877.
- [29] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. USENIX Association, 15–28.
- [30] Wangda Zhang, Junyoung Kim, Kenneth A. Ross, Eric Sedlar, and Lukas Stadler. 2021. Adaptive Code Generation for Data-Intensive Analytics. *Proc. VLDB Endow.* 14, 6 (2021), 929–942.
- [31] Yuhao Zhang, Frank McQuillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. 2021. Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches. *Proc. VLDB Endow.* 14, 10 (2021), 1769–1782.
- [32] Jia Zou, Amitabh Das, Pratik Barhate, Arun Iyengar, Binhang Yuan, Dimitrije Jankov, and Chris Jermaine. 2021. Lachesis: Automated Partitioning for UDF-Centric Analytics. *Proc. VLDB Endow.* 14, 8 (2021), 1262–1275.
- [33] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. 2012. Vectorwise: A Vectorized Analytical DBMS. In *ICDE*. IEEE Computer Society, 1349–1350.