# DuoLingo-AutoDiff

## In-Database Automatic Differentiation with MLIR

Kevin Gutjahr
University of Bamberg
kevin.gutjahr@stud.uni-bamberg.de

Clemens Ruck
University of Bamberg
clemens.ruck@uni-bamberg.de

Maximilian E. Schüle
University of Bamberg
maximilian.schuele@uni-bamberg.de

## Abstract

Forward and reverse mode automatic differentiation evaluate the gradient of a model function efficiently by caching the results of partial derivatives. Just-in-time compilation improves the runtime of automatic differentiation by eliminating function calls and storing partial derivatives in virtual registers. This paper discusses the first open-source implementation of automatic differentiation with MLIR and LingoDB. The evaluation compares optimizations applied to forward and reverse modes. It showed that sub-expressions, that appear frequently within the calculation, will be reused after MLIR performs its optimization. Additionally, reverse mode outperforms forward mode due to less generated code.

## Keywords

Automatic Differentiation, In-Database Machine Learning, Query Compilation, MLIR

## 1 Introduction

In-database machine learning (ML) allows training algorithms to access the latest database state [9, 10]. This eliminates the need for costly data extraction required for end-to-end machine learning pipelines [3, 4, 19, 27]. In addition, code-generating database systems provide a compiler infrastructure that can be used to accelerate certain parts within machine learning pipelines [15]. For example, gradient descent training requires automatically derived model functions. Automatic differentiation applies the chain rule to partially derive a function either from the inner to the outer partial expression (forward mode) or from the outer to the inner (reverse mode). In both forward and reverse mode automatic differentiation, partial derivatives are cached when computing the gradient of a model function. In the past, we have shown that code generation with a low-level virtual machine (LLVM) assembler accelerates the evaluation of the generated gradients by storing the partial results as virtual registers.

This work ports automatic differentiation to another compiler infrastructure, Multi-Level Intermediate Representation (MLIR): MLIR is a compiler infrastructure that facilitates the definition of domain-specific dialects of intermediate code representations, creating a flexible and extensible framework. The purpose of these representations is to progressively lower the given code towards LLVM [13] code. In the case of LingoDB—the database system selected for this work—MLIR lowers the user's SQL query, thereby enabling efficient execution and optimization for the underlying hardware. The objective of this work is to analyze the code generated by both forward and reverse mode implemented in LingoDB[1] and how it is optimized by MLIR afterwards.

This paper is structured as follows: Section 2 explains the algorithms of both automatic differentiation modes in greater detail, while Section 3 does so for LingoDB. Section 4 presents the implementation of the new operators in the database system. Section 5 compares the performances of the two modes and their generated code. Section 6 concludes this work.

## 2 Automatic Differentiation

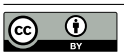
Besides automatic differentiation, there is symbolic differentiation and numerical differentiation [2]. However, symbolic differentiation manipulates the equation directly by replacing each expression with its derivative. While this leads to accurate results, it also leads to *expression swell*. The transformation can lead to an exponential growth of expressions, as the applied differentiation rules can lead to nested transformations, which also need to be evaluated [2]. Numerical differentiation is not ideal either, since it only computes numerical approximations of the partial derivatives [2]. Besides floating-point truncation errors, imprecise derivatives also undermine model training, as they can bias the weights in a direction other than the desired, optimal state.

In automatic differentiation, the partial derivatives are computed by reconstructing the equation into a computational graph, where each node represents an individual expression of the equation [16]. Leaf nodes represent parameters and constants, while all other nodes correspond to arithmetic operations such as addition and subtraction. The root nodes of the graph represent the results. The partial derivatives are then computed by traversing this graph and applying the chain rule to each node. Automatic differentiation can be implemented using two approaches: forward mode and reverse mode.

### 2.1 Forward Mode

An example of the forward mode accumulation in automatic differentiation is illustrated in Fig. 1. It traverses through the constructed expression tree once for every variable used in the equation [16]. At each run, it starts at the leaf nodes and moves up in the tree towards the root node. If the current leaf node corresponds to the parameter with respect to which the partial derivative is being computed in this run, its intermediate partial derivative is set to 1. If it corresponds to another parameter, it is set to 0. As the traversal moves up, the chain rule is applied to compute the partial derivative at each node using the derivatives of its child nodes. This also requires evaluating the equation up to the current node.

[1] https://github.com/MaxEmanuel/lingo-db




*DEEM '25, Berlin, Germany*

ACM ISBN 979-8-4007-1924-0/2025/06
https://doi.org/10.1145/3735654.3735943

**Algorithm 1** Forward Mode [25]

```
1:  function DERIVATE(X, V)
2:      if isVariable(X) then
3:          if X == V then
4:              return 1
5:          else
6:              return 0
7:      else
8:          A' ← derivate(A, V), B' ← derivate(B, V)
9:          if X = A + B then
10:             return A' + B'
11:         else if X = A − B then
12:             return A' − B'
13:         else if X = A · B then
14:             return A' · B + B' · A
15:         else if X = A/B then
16:             return (A'·B − B'·A)/B²
```

Since this algorithm runs once per parameter, it is more suitable for functions $f : \mathbb{R}^n \to \mathbb{R}^m$, where $n < m$. Algorithm 1 illustrates the procedure for the four fundamental arithmetic operations.

## 2.2 Reverse Mode

Unlike forward mode, reverse mode starts at the root node and traverses the computational graph downward to the leaf nodes [16]. This is illustrated in Fig. 2. The algorithm computes the partial derivative of the parent node with respect to the current node. At the root node, the algorithm starts with a seed value of 1, since the derivative of the output with respect to itself is 1. This serves as the initialization step for the differentiation process. As the algorithm moves down the graph, it applies the chain rule at each step to compute the partial derivatives at the individual nodes. This process requires not only the computation of derivatives but also the evaluation of the numerical values of the nodes themselves, as these values are necessary for correctly applying the chain rule.

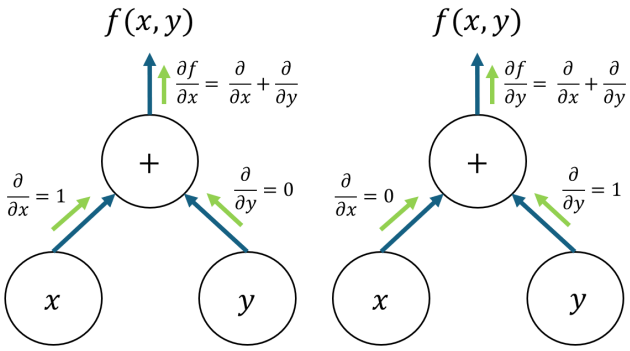



**Figure 1: Forward mode accumulation visualized for the equation $f(x, y) = x + y$. The algorithm analyzes the equation once for each variable, in this case twice: once for $x$ and once for $y$ [16]. The green arrows indicate the direction of the algorithm's flow.**

**Algorithm 2** Reverse Mode [25]

```
1:  function DERIVATE(X, seed)
2:      if X = A + B then
3:          derivate(A, seed), derivate(B, seed)
4:      else if X = A − B then
5:          derivate(A, seed), derivate(B, −seed)
6:      else if X = A · B then
7:          derivate(A, B · seed), derivate(B, A · seed)
8:      else if X = A/B then
9:          derivate(A, seed/B), derivate(B, −seed · A/B²)
10:     else if isVariable(X) then
11:         X' ← X' + seed
12:         return X'
```

Therefore, the algorithm has to traverse the graph twice, once to obtain the numerical values of the nodes and once to calculate the partial derivatives. Algorithm 2 demonstrates the process for performing the four fundamental arithmetic operations. As this algorithm is run once for every root node, it is more suitable for functions $f : \mathbb{R}^n \to \mathbb{R}^m$, where $n > m$. This characteristic makes reverse mode particularly suitable for use in machine learning, as the dimensions of the gradients required are rather small compared to the amount of input [2].

# 3 LingoDB

The system used in this thesis is LingoDB [12], a relational database system that uses compiler technology. The system translates SQL queries into LLVM code [13], facilitating efficient execution of user queries. The primary goal of LingoDB is to provide a system that allows new features to be added with relatively little implementation effort, while maintaining the performance of today's state-of-the-art database systems.

## 3.1 Compiler Infrastructure

To achieve this, LingoDB was developed using the MLIR compiler infrastructure [14], which is specifically designed to support the creation of domain-specific compilers. To provide a high degree of

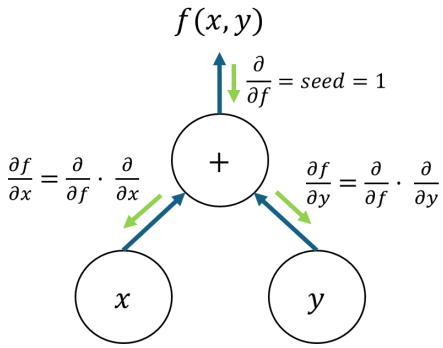



**Figure 2: Reverse mode accumulation visualized for the equation $f(x, y) = x + y$. Here, the equation is analyzed only once as it has only one root node [16].**

extensibility, MLIR introduces a new, modular Intermediate Representation (IR). An IR is a temporary encoding of the input source code and a key component of a compiler [1].

It simplifies the translation of high-level source code into low-level machine code for different target architectures by acting as an additional abstraction layer in between. The general structure of a compiler is illustrated in Fig. 3. The compiler's front-end generates an IR by parsing the source code into its syntactical components and then reorganizing them. During this process, the code is also grammatically checked according to the rules of the source language. After the IR code has been generated, it undergoes several optimization steps in most compilers. The code is then processed by the compiler's back end to produce machine code tailored to the target processor. An IR can be structured in a variety of ways, including linear (a sequence of instructions) or graphical (a tree representing relationships within the program) [26]. Different implementations offer different advantages and disadvantages, making them suitable for different use cases. This often leads developers to create their own IR to solve their domain-specific problems, despite the high development effort required for a new IR [14]. This also leads to re-implementation of existing solutions for common problems, resulting in unnecessary effort for the developer. MLIR addresses this problem by modularizing its IR [14]. The syntax is built from so-called *operations*, which are grouped into dialects.

Operations represent all the actions that a computer program can perform, such as function calls and variable assignments. These elements produce at least one result, called values, which have predefined types, such as integers or floats. In addition, operations can take other values as input. These values are evaluated at runtime and are managed in static single assignment (SSA) form. In SSA form, each assignment to a variable is treated as a separate assignment to a uniquely versioned variable [5]. An illustration of this is given in Fig. 4. Here, two consecutive assignments to one variable are converted to two variables with one assignment each. This helps the compiler perform optimization tasks such as dead code detection, as it can more easily identify which instructions affect the result. In addition, operations can include regions consisting of one or more blocks. Blocks act as containers for operations and define the execution flow within a region. An example of the resulting IR structure is shown in Listing 1. These semantics allow developers to easily introduce new operations or even entirely new dialects into

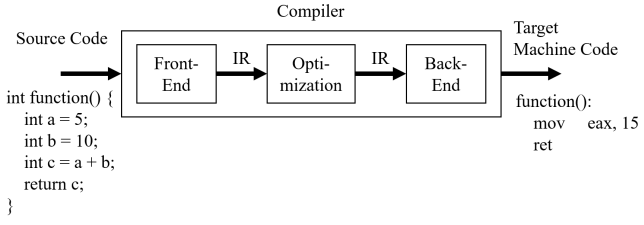



**Figure 3: Simplified structure of a compiler. The Intermediate Representation of the original source code acts as a layer between the high-level source code and the low-level machine code [1]. This approach also improves compiler modularity, which in turn increases their maintainability.**

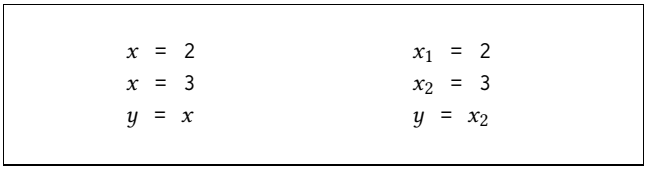



**Figure 4: An example of the SSA form [5]. The original code is on the left, while the SSA form of it is on the right. The second assignment to $x$ is treated as a assignment to a new, unique variable, $x_2$. Now, it is evident to the optimizer that the first assignment to $x$ is redundant.**

the infrastructure, allowing them to reuse existing components and reduce implementation effort. Using this infrastructure, LingoDB has introduced new dialects to enable database functionality such as relational algebra and additional data types such as timestamp, datetime, strings, and numeric types [11, 12].

## 3.2 System Architecture

The main components of LingoDB are illustrated in Fig. 5. They consist of a front-end, an MLIR optimization step, the lowering of the optimized MLIR code, the compilation of the resulting LLVM code, and a runtime system [12]. The front-end generates an MLIR representation of the incoming SQL Query. The program breaks the query down into its individual components using lexical and grammatical analysis. This is implemented using GNU Bison [8], a parser generator. By processing a provided list of possible SQL keywords and the query language's grammar definition, Bison generates C code that constructs a syntax tree from the input query. Each node in this tree represents a query element, such as a keyword or a constant. It then traverses this tree and produces, according to the current node, fitting MLIR operations. In the next step, several query optimization steps are performed on the resulting IR code

```
1  module {
2   func.func @main() {
3    %0 = relalg.const_relation columns : [@dummyScope::@dummyName({type = i32})]
          values : [[0]]
4    %1 = relalg.map %0 computes : [@map0::@tmp_attr0({type = i32})] (%arg0: !
          tuples.tuple){
5     %3 = db.constant(15 : i32) : i32
6     tuples.return %3 : i32
7    }
8    %2 = relalg.materialize %1 [@map0::@tmp_attr0] => [""] : !subop.result_table<[
          unnamed$0 : i32]>
9    subop.set_result 0 %2 : !subop.result_table<[unnamed$0 : i32]>
10   return
11  }
12 }
```

**Listing 1: The optimized MLIR code resulting from the query "SELECT 5 + 10;". At line 3, a new, empty relation is created, while at line 4, a mapping operation is performed to add the result to it. The calculation takes place within the region of relalg.map. In this case, 'relalg' refers to the name of the dialect, while 'map' is the name of the operation. Due to optimization, it is already reduced to simply returning the final result, 15. At line 8, the filled relation is materialized, which will be returned at line 9 and 10.**

[12]. First, it utilizes pre-built MLIR passes, which perform optimizations such as removal of dead code, common sub-expressions (CSE), and canonicalization of dialects [14]. This step simplifies the code and facilitates further optimization steps. Next, it optimizes the underlying execution plan [12]. It begins by un-nesting relational algebra operations. Nested operations, such as sub-queries, lead to performance discrepancies as they have to be re-evaluated multiple times during the query execution [18]. Subsequently, selections are pushed down as far as possible so that the data is as pre-processed before selection as possible. Afterward, the query's join order is optimized. In the last optimization step, frequently used tuples are materialized to avoid re-evaluation. Once the optimization is completed, the resulting MLIR code is lowered in several steps, from high-level dialects such as *relalg* (relational algebra operations) to low-level dialects like *llvm*. In order to lower the complex, relational algebra operations, LingoDB utilizes data-centric code generation [17]. The goal of this approach is to keep the data as long in the CPU's register as possible, avoiding unnecessary movement between the CPU and the main memory. To achieve this, data-centric code generation introduces the *consume-produce* concept. Here, the operator's interface consists of two functionalities, *consume()* and *produce(). produce()* tells the operator to create its results, while *consume()* is used to receive incoming data. Operators invoke the produce function of their child operations recursively in order to receive data, process it, and then call the consume function of their parent operation to continue processing. The required data is loaded into the CPU's register first and then successively processed by as many operators as possible. Using this concept, LingoDB lowers *relalg* operations to *db* operations (database-specific operations and data-types) [12]. The resulting MLIR code will describe the required workflow step by step. The remaining dialects are progressively lowered in multiple steps until only the *llvm* dialect remains. The resulting LLVM-Code will then be compiled just in time (JIT) to produce system-dependent machine code. During this process, additional optimization passes are performed to further enhance the machine code's efficiency. Finally, this code will then be executed on the runtime system which is built on Apache Arrow [7]. This is a database format, which runs on main memory and allows fast data access. At runtime, LingoDB can call C++ functions to perform more complex operations, such as sorting algorithms.

## 4 Implementation

Both backward and forward mode automatic differentiation were integrated into LingoDB. For this purpose, a total of four SQL operators had to be added to LingoDB's front-end: `TABLE`, `LAMBDA`, `DERIVATEBACKWARDS`, and `DERIVATEFORWARDS`. `TABLE` is a new keyword that creates a new, temporarily available table.

```
TABLE(<table-reference>); TABLE(<SQL-Subquery>).
```

It can either accept a reference to an already existing table or a sub-query as its input. In both cases, it processes the provided query and stores the result as a new common table expression (CTE). A CTE is a temporary table that resides in memory and remains accessible only within the scope of the current query execution. By utilizing CTEs in this manner, the operator allows for the definition of input variable data needed for the derivative functionality without requiring the creation of a new, persistent table. The newly generated CTE will be assigned a unique name, allowing it to be referenced by other elements within the query.

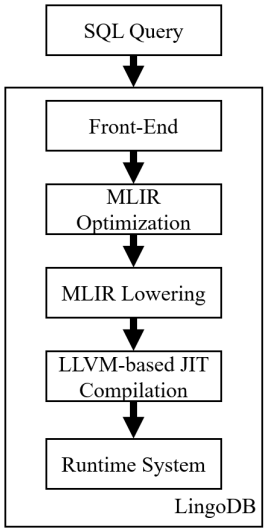



**Figure 5: Key components of LingoDB [12]. The input SQL Query is translated into MLIR code, which is then optimized and lowered. The resulting LLVM code is then compiled into processor dependent machine code, which is then executed in the runtime system.**

The `LAMBDA` can be used to define SQL expressions as mathematical equations [20–24]:

```
LAMBDA{<table1>, <table2>, ···}(<SQL-Expression>).
```

Within the curly brackets, the operator takes references to input tables, with their columns being used as variables in the mathematical equation. In the round brackets, it takes an SQL expression. The expression will be parsed into a syntax tree by the front-end. When evaluating the equation, the front-end will generate MLIR code that represents the structure of the tree.

`DERIVATEBACKWARDS` and `DERIVATEFORWARDS` have been implemented as new table functions:

```
DERIVATEBACKWARDS(<TABLE(···)>, <TABLE(···)>, ···,
    <LAMBDA{···}(···)>).

DERIVATEFORWARDS(<TABLE(···)>, <TABLE(···)>, ···, <
    LAMBDA{···}(···)>).
```

Both functions accept a `LAMBDA` expression, which must be the final argument in the parameter list. Additionally, they can also take `TABLE` expressions as parameters for additional input variable definition. The front-end will generate MLIR code for differentiating the function body of the `LAMBDA` expression, which will then be executed on all tuples from the input tables. For this purpose, the input tables will be temporarily concatenated into a single table. The resulting partial derivatives are then appended to that table. Before evaluation, `DERIVATEFORWARDS` must traverse the equation tree of the `LAMBDA` expression to extract a list of all

variables that are actually used. It will then apply the automatic differentiation algorithm in forward mode for each identified variable. Using `DERIVATEBACKWARDS`, the algorithm will be executed once in reverse mode. The supported arithmetic operations are addition, subtraction, multiplication, division, and exponentiation. Additionally, the functions square root, exponential, logarithm, sine, and cosine can also be derived.

## 5 Experiments

Two experiments were conducted on a system with an Intel Xeon W-2295 processor (18 cores, 24.75 MB Cache, 3.00 GHz) and 125 GB of RAM. The first experiment compares the MLIR code of two differentiation algorithms and analyzes how it is optimized by MLIR. The second experiment compares the algorithms' performance in timing under a higher workload. The processing time is split into three parts: optimization time (the time taken for query optimization on the MLIR code), compilation time (the lowering passes to generate low-level MLIR in the *llvm* dialect), and execution time (the time to execute the generated machine code).

### 5.1 Code Generation

The performance of the system depends on the code generated during query processing. To see how the differing algorithms affect the code, the query seen in Listing 2 was used. Here, it calculates the partial derivatives of the loss function Root Mean Squared Logarithmic Error (RSMLE), which is used for the linear model $x_0 \cdot w_0 + x_1 \cdot w_1 + b$:

$$L = \sqrt{\left(\log\left(x_0 \cdot w_0 + x_1 \cdot w_1 + b + 1\right) - \log\left(y_0 + 1\right)\right)^2}$$

As this function contains nested expressions, it is also a good candidate to analyze MLIR's optimization. The MLIR code generated by the reverse mode is displayed in Fig. 6 (unoptimized) and Fig. 7 (optimized). The figures present the code needed to calculate the partial derivative of $x_0$. A comparison reveals that MLIR optimizes the code by removing common sub-expressions. In the unoptimized code, the partial derivative is calculated as

$$\begin{aligned}\frac{\partial L}{\partial x_0} = 1 \cdot 0.5 \cdot & \frac{1}{\sqrt{\left(\log\left(x_0 \cdot w_0 + x_1 \cdot w_1 + b + 1\right) - \log\left(y_0 + 1\right)\right)^2}} \\ & \cdot 2 \cdot \left(\log\left(x_0 \cdot w_0 + x_1 \cdot w_1 + b + 1\right) - \log\left(y_0 + 1\right)\right)^{(2-1)} \\ & \cdot \frac{1}{x_0 \cdot w_0 + x_1 \cdot w_1 + b + 1} \cdot w_0.\end{aligned}$$

Here, the sub-expressions $x_0 \cdot w_0 + x_1 \cdot w_1 + b + 1$ and $y_0 + 1$ are used at multiple locations. In this example, this redundancy takes up the majority of the generated code. Since LingoDB utilizes MLIR's CSE pass in the optimization step, it recognizes these sub-expressions and rewrites the code so that they are calculated first. Their results are then reused at the corresponding locations in the equation. The same applies to constants and column reads. $\frac{\partial L}{\partial x_0}$ is then calculated as

$$\begin{aligned}c &= x_0 \cdot w_0 + x_1 \cdot w_1 + b + 1 \\ d &= y_0 + 1 \\ \frac{\partial L}{\partial x_0} &= 1 \cdot 0.5 \cdot \frac{1}{\sqrt{\left(\log\left(c\right) - \log\left(d\right)\right)^2}} \\ & \quad \cdot 2 \cdot \left(\log\left(c\right) - \log\left(d\right)\right)^{(2-1)} \cdot \frac{1}{c} \cdot w_0.\end{aligned}$$

This enables the system to simplify the code and cut its size in half. However, it doesn't recognize the reuse of the logarithmic expressions that take these sub-expressions as inputs and still recalculates them.

This behavior can also be observed in the forward mode, as demonstrated in Fig. 8. Here, MLIR recognizes the reuse of the same sub-expressions, namely $x_0 \cdot w_0 + x_1 \cdot w_1 + b + 1$ and $y_0 + 1$, as in reverse mode. However, the forward mode code is considerably longer than the code of the reverse mode. In forward mode, the algorithm traverses the equation tree from the leaf nodes up to the root nodes. If the current leaf node represents the variable for which the partial derivative is being calculated, the algorithm returns 1 as the current partial derivative. If the node does not represent it, the algorithm returns 0. The system generates the required expressions of Algorithm 1 at every node in the tree. However, since MLIR evaluates the generated code only at runtime, it does not simplify the code by removing expressions that are subsequently equal to 0. This leads $\frac{\partial L}{\partial x_0}$ to be calculated as

$$\begin{aligned}c &= x_0 \cdot w_0 + x_1 \cdot w_1 + b + 1 \\ d &= y_0 + 1 \\ \frac{\partial L}{\partial x_0} &= 0.5 \cdot \frac{1}{\sqrt{\left(\log\left(c\right) - \log\left(d\right)\right)^2}} \\ & \quad \cdot \Big( \Big( \big( 0 \cdot \left(\log\left(c\right) - \log\left(d\right)\right)^2 \cdot \log\left(2\right) \big) \Big( \\ & \quad + \quad \left(2 \cdot \left(\log\left(c\right) - \log\left(d\right)\right)^{(2-1)}\right) \Big( \\ & \quad \cdot \quad \left( (0 + 0 + x_0 \cdot 0 + w_0 \cdot 1 + x_1 \cdot 0 + w_1 \cdot 0) \cdot \frac{1}{c} \right) \\ & \quad - \Big( \Big( \big( 0 + 0 \big) \cdot \frac{1}{d} \Big) \Big) \Big) \Big) \Big(.\end{aligned}$$

Fig. 9 shows that this results in a decline in performance when compared to the reverse mode. The graph illustrates the mean processing times for executing Listing 2, which was repeated 1,000 times. The used test relation contained 75000 tuples of dummy data.

```
select * from derivateBackwards(TABLE(test), TABLE(select 1.0 as y0), TABLE(
      select random() as w0, random() as w1, random() as b), lambda{test, x, x1
      }(sqrt((log(test.x0 * x1.w0 + test.x1 * x1.w1 + x1.b + 1) - log(x.y0 + 1)
      )^2)));
```

**Listing 2: Calculating the derivatives of the RSMLE loss function using our SQL functions. It is used twice: once as DerivateForwards and once as DerivateBackwards.**

```
%13 = db.constant(1.000000e+00 : f64) : f64
%14 = db.constant(5.000000e-01 : f64) : f64
%15 = db.constant(1.000000e+00 : f64) : f64
%16 = tuples.getcol %arg0 @test1::@x0 : !db.decimal<2, 1>
%17 = tuples.getcol %arg0 @x1::@w0 : !db.decimal<3, 2>
%18 = db.mul %16 : !db.decimal<2, 1>, %17 : !db.decimal<3, 2>
%19 = tuples.getcol %arg0 @test1::@x1 : !db.decimal<2, 1>
%20 = tuples.getcol %arg0 @x1::@w1 : !db.decimal<3, 2>
%21 = db.mul %19 : !db.decimal<2, 1>, %20 : !db.decimal<3, 2>
%22 = db.add %18 : !db.decimal<5, 3>, %21 : !db.decimal<5, 3>
%23 = tuples.getcol %arg0 @x1::@b : !db.decimal<3, 2>
%24 = db.cast %23 : !db.decimal<3, 2> -> !db.decimal<5, 3>
%25 = db.add %22 : !db.decimal<5, 3>, %24 : !db.decimal<5, 3>
%26 = db.constant(1 : i32) : !db.decimal<5, 3>
%27 = db.add %25 : !db.decimal<5, 3>, %26 : !db.decimal<5, 3>
%28 = db.cast %27 : !db.decimal<5, 3> -> f64
%29 = db.runtime_call "Log"(%28) : (f64) -> f64
%30 = tuples.getcol %arg0 @x::@y0 : !db.decimal<2, 1>
%31 = db.constant(1 : i32) : !db.decimal<2, 1>
%32 = db.add %30 : !db.decimal<2, 1>, %31 : !db.decimal<2, 1>
%33 = db.cast %32 : !db.decimal<2, 1> -> f64
%34 = db.runtime_call "Log"(%33) : (f64) -> f64
%35 = db.sub %29 : f64, %34 : f64
%36 = db.constant(2 : i32) : f64
%37 = db.runtime_call "PowerFloat"(%35, %36) : (f64, f64) -> f64
%38 = db.runtime_call "Sqrt"(%37) : (f64) -> f64
%39 = db.div %15 : f64, %38 : f64
%40 = db.mul %14 : f64, %39 : f64
%41 = db.mul %13 : f64, %40 : f64
%42 = tuples.getcol %arg0 @test1::@x0 : !db.decimal<2, 1>
%43 = tuples.getcol %arg0 @x1::@w0 : !db.decimal<3, 2>
%44 = db.mul %42 : !db.decimal<2, 1>, %43 : !db.decimal<3, 2>
%45 = tuples.getcol %arg0 @test1::@x1 : !db.decimal<2, 1>
%46 = tuples.getcol %arg0 @x1::@w1 : !db.decimal<3, 2>
%47 = db.mul %45 : !db.decimal<2, 1>, %46 : !db.decimal<3, 2>
%48 = db.add %44 : !db.decimal<5, 3>, %47 : !db.decimal<5, 3>
%49 = tuples.getcol %arg0 @x1::@b : !db.decimal<3, 2>
%50 = db.cast %49 : !db.decimal<3, 2> -> !db.decimal<5, 3>
%51 = db.add %48 : !db.decimal<5, 3>, %50 : !db.decimal<5, 3>
%52 = db.constant(1 : i32) : !db.decimal<5, 3>
%53 = db.add %51 : !db.decimal<5, 3>, %52 : !db.decimal<5, 3>
%54 = db.cast %53 : !db.decimal<5, 3> -> f64
%55 = db.runtime_call "Log"(%54) : (f64) -> f64
%56 = tuples.getcol %arg0 @x::@y0 : !db.decimal<2, 1>
%57 = db.constant(1 : i32) : !db.decimal<2, 1>
%58 = db.add %56 : !db.decimal<2, 1>, %57 : !db.decimal<2, 1>
%59 = db.cast %58 : !db.decimal<2, 1> -> f64
%60 = db.runtime_call "Log"(%59) : (f64) -> f64
%61 = db.sub %55 : f64, %60 : f64
%62 = db.constant(2 : i32) : f64
%63 = db.constant(1.000000e+00 : f64) : f64
%64 = db.sub %62 : f64, %63 : f64
%65 = db.runtime_call "PowerFloat"(%61, %64) : (f64, f64) -> f64
%68 = db.mul %62 : f64, %65 : f64
%69 = db.mul %41 : f64, %68 : f64
%70 = db.constant(1.000000e+00 : f64) : f64
%71 = tuples.getcol %arg0 @test1::@x0 : !db.decimal<2, 1>
%72 = tuples.getcol %arg0 @x1::@w0 : !db.decimal<3, 2>
%73 = db.mul %71 : !db.decimal<2, 1>, %72 : !db.decimal<3, 2>
%74 = tuples.getcol %arg0 @test1::@x1 : !db.decimal<2, 1>
%75 = tuples.getcol %arg0 @x1::@w1 : !db.decimal<3, 2>
%76 = db.mul %74 : !db.decimal<2, 1>, %75 : !db.decimal<3, 2>
%77 = db.add %73 : !db.decimal<5, 3>, %76 : !db.decimal<5, 3>
%78 = tuples.getcol %arg0 @x1::@b : !db.decimal<3, 2>
%79 = db.cast %78 : !db.decimal<3, 2> -> !db.decimal<5, 3>
%80 = db.add %77 : !db.decimal<5, 3>, %79 : !db.decimal<5, 3>
%81 = db.constant(1 : i32) : !db.decimal<5, 3>
%82 = db.add %80 : !db.decimal<5, 3>, %81 : !db.decimal<5, 3>
%83 = db.cast %82 : !db.decimal<5, 3> -> f64
%84 = db.div %70 : f64, %83 : f64
%85 = db.mul %69 : f64, %84 : f64
%86 = tuples.getcol %arg0 @x1::@w0 : !db.decimal<3, 2>
%87 = db.cast %86 : !db.decimal<3, 2> -> f64
%88 = db.mul %85 : f64, %87 : f64
```

**Figure 6: MLIR code generated by reverse mode before optimization is applied. The code re-calculates the sub-expression by reading the same constants and columns (red) and executing the same instructions (blue) multiple times.**

When using forward mode, the system requires more time during optimization and compilation because it has to process more code compared to reverse mode. However, the execution times of both algorithms are nearly identical. Forward mode takes 150.95 ms for execution, 65.83 ms for optimization, and 145.32 ms for compilation (362.1 ms in total). Meanwhile, reverse mode takes 133.99 ms for execution, 35.23 ms for optimization, and 105.37 ms for compilation (274.59 ms in total).

```
%21 = db.constant(2 : i32) : f64
%22 = db.constant(1 : i32) : !db.decimal<2, 1>
%23 = db.constant(1 : i32) : !db.decimal<5, 3>
%24 = db.constant(1.000000e+00 : f64) : f64
%25 = db.constant(5.000000e-01 : f64) : f64
%26 = tuples.getcol %arg0 @test1::@x0 : !db.decimal<2, 1>
%27 = tuples.getcol %arg0 @x1::@w0 : !db.decimal<3, 2>
%28 = db.mul %26 : !db.decimal<2, 1>, %27 : !db.decimal<3, 2>
%29 = tuples.getcol %arg0 @test1::@x1 : !db.decimal<2, 1>
%30 = tuples.getcol %arg0 @x1::@w1 : !db.decimal<3, 2>
%31 = db.mul %29 : !db.decimal<2, 1>, %30 : !db.decimal<3, 2>
%32 = db.add %28 : !db.decimal<5, 3>, %31 : !db.decimal<5, 3>
%33 = tuples.getcol %arg0 @x1::@b : !db.decimal<3, 2>
%34 = db.cast %33 : !db.decimal<3, 2> -> !db.decimal<5, 3>
%35 = db.add %32 : !db.decimal<5, 3>, %34 : !db.decimal<5, 3>
%36 = db.add %35 : !db.decimal<5, 3>, %23 : !db.decimal<5, 3>
%37 = db.cast %36 : !db.decimal<5, 3> -> f64
%38 = db.runtime_call "Log"(%37) : (f64) -> f64
%39 = tuples.getcol %arg0 @x::@y0 : !db.decimal<2, 1>
%40 = db.add %39 : !db.decimal<2, 1>, %22 : !db.decimal<2, 1>
%41 = db.cast %40 : !db.decimal<2, 1> -> f64
%42 = db.runtime_call "Log"(%41) : (f64) -> f64
%43 = db.sub %38 : f64, %42 : f64
%44 = db.runtime_call "PowerFloat"(%43, %21) : (f64, f64) -> f64
%45 = db.runtime_call "Sqrt"(%44) : (f64) -> f64
%46 = db.div %24 : f64, %45 : f64
%47 = db.mul %25 : f64, %46 : f64
%48 = db.mul %24 : f64, %47 : f64
%49 = db.runtime_call "Log"(%37) : (f64) -> f64
%50 = db.runtime_call "Log"(%41) : (f64) -> f64
%51 = db.sub %49 : f64, %50 : f64
%52 = db.sub %21 : f64, %24 : f64
%53 = db.runtime_call "PowerFloat"(%51, %52) : (f64, f64) -> f64
%54 = db.mul %21 : f64, %53 : f64
%55 = db.mul %48 : f64, %54 : f64
%56 = db.div %24 : f64, %37 : f64
%57 = db.mul %55 : f64, %56 : f64
%58 = db.cast %27 : !db.decimal<3, 2> -> f64
%59 = db.mul %57 : f64, %58 : f64
```

**Figure 7: Optimized reverse mode MLIR code. The code calculates the sub-expression only once and reuses the result where it is needed (brown).**

## 5.2 Performance

The second experiment is meant to test both algorithms under a higher workload than in the first experiment. Listing 3 shows the query used to achieve this. It calculates the partial derivatives of

$$f(s_l, s_w, p_l, p_w, w_1, w_2, w_3, w_4, b) = \sigma(s_l \cdot w_1 + s_w \cdot w_2 + p_l \cdot w_3 + p_w \cdot w_4 + b).$$

This equation represents a neural network layer that receives four input values and produces a single output with the sigmoid activation function applied to it. The query was run on the iris dataset [6], which contains 150 tuples representing flowers. Each tuple consists of measurements of the flowers, namely sepal length $s_l$, sepal width $s_w$, petal length $p_l$, petal width $p_w$, and the species of the flower. For this experiment, the data set was multiplied by 2000 to create 300000 tuples. As in the first experiment, the queries are executed 1000 times to reduce random variation in the measured processing times. The results are shown in Fig. 10. Similar to the results in Section 5.1, there is a difference between the processing times of the two differentiation modes. Again, the forward mode takes more time than the reverse mode. The forward mode took on average 76.68 ms to complete its optimization, 145.38 ms to compile and 626.72 ms to execute. Meanwhile, reverse mode took an average of 40.28 ms for optimization, 108.73 ms for compilation, and 629.67 ms for execution.

This results in a total processing time of 848.78 ms using the forward mode algorithm and 778.68 ms using the reverse mode algorithm, a difference of ~70 ms. The reason for this behavior can be found in the generated MLIR code. As explained in Section 5.1,

```
%21 = db.constant(5.000000e-01 : f64) : f64
%22 = db.constant(2 : i32) : f64
%23 = db.constant(1 : i32) : !db.decimal<2, 1>
%24 = db.constant(1 : i32) : !db.decimal<5, 3>
%25 = db.constant(1.000000e+00 : f64) : f64
%26 = db.constant(0.000000e+00 : f64) : f64
%27 = tuples.getcol %arg0 @test1::@x0 : !db.decimal<2, 1>
%28 = db.cast %27 : !db.decimal<2, 1> -> f64
%29 = db.mul %28 : f64, %26 : f64
%30 = tuples.getcol %arg0 @x1::@w0 : !db.decimal<3, 2>
%31 = db.cast %30 : !db.decimal<3, 2> -> f64
%32 = db.mul %31 : f64, %25 : f64
%33 = db.add %29 : f64, %32 : f64
%34 = tuples.getcol %arg0 @test1::@x1 : !db.decimal<2, 1>
%35 = db.cast %34 : !db.decimal<2, 1> -> f64
%36 = db.mul %35 : f64, %26 : f64
%37 = tuples.getcol %arg0 @x1::@w1 : !db.decimal<3, 2>
%38 = db.cast %37 : !db.decimal<3, 2> -> f64
%39 = db.mul %38 : f64, %26 : f64
%40 = db.add %36 : f64, %39 : f64
%41 = db.add %33 : f64, %40 : f64
%42 = db.add %41 : f64, %26 : f64
%43 = db.add %42 : f64, %26 : f64
%44 = db.mul %27 : !db.decimal<2, 1>, %30 : !db.decimal<3, 2>
%45 = db.mul %34 : !db.decimal<2, 1>, %37 : !db.decimal<3, 2>
%46 = db.add %44 : !db.decimal<5, 3>, %45 : !db.decimal<5, 3>
%47 = tuples.getcol %arg0 @x1::@b : !db.decimal<3, 2>
%48 = db.cast %47 : !db.decimal<3, 2> -> !db.decimal<5, 3>
%49 = db.add %46 : !db.decimal<5, 3>, %48 : !db.decimal<5, 3>
%50 = db.add %49 : !db.decimal<5, 3>, %24 : !db.decimal<5, 3>
%51 = db.cast %50 : !db.decimal<5, 3> -> f64
%52 = db.div %25 : f64, %51 : f64
%53 = db.mul %43 : f64, %52 : f64
%54 = db.add %26 : f64, %26 : f64
%55 = tuples.getcol %arg0 @x::@y0 : !db.decimal<2, 1>
%56 = db.add %55 : !db.decimal<2, 1>, %23 : !db.decimal<2, 1>
%57 = db.cast %56 : !db.decimal<2, 1> -> f64
%58 = db.div %25 : f64, %57 : f64
%59 = db.mul %54 : f64, %58 : f64
%60 = db.sub %53 : f64, %59 : f64
%61 = db.runtime_call "Log"(%51) : (f64) -> f64
%62 = db.runtime_call "Log"(%57) : (f64) -> f64
%63 = db.sub %61 : f64, %62 : f64
%64 = db.sub %22 : f64, %25 : f64
%65 = db.runtime_call "PowerFloat"(%63, %64) : (f64, f64) -> f64
%66 = db.mul %22 : f64, %65 : f64
%67 = db.mul %60 : f64, %66 : f64
%68 = db.runtime_call "Log"(%51) : (f64) -> f64
%69 = db.runtime_call "Log"(%57) : (f64) -> f64
%70 = db.sub %68 : f64, %69 : f64
%71 = db.runtime_call "PowerFloat"(%70, %22) : (f64, f64) -> f64
%72 = db.runtime_call "Log"(%22) : (f64) -> f64
%73 = db.mul %71 : f64, %72 : f64
%74 = db.mul %26 : f64, %73 : f64
%75 = db.add %67 : f64, %74 : f64
%76 = db.runtime_call "Log"(%51) : (f64) -> f64
%77 = db.runtime_call "Log"(%57) : (f64) -> f64
%78 = db.sub %76 : f64, %77 : f64
%79 = db.runtime_call "PowerFloat"(%78, %22) : (f64, f64) -> f64
%80 = db.runtime_call "Sqrt"(%79) : (f64) -> f64
%81 = db.div %25 : f64, %80 : f64
%82 = db.mul %21 : f64, %81 : f64
%83 = db.mul %75 : f64, %82 : f64
```

**Figure 8: Optimized forward mode MLIR code. The same behavior as in the reverse mode code can be seen.**

MLIR only evaluates the generated code at runtime and therefore doesn't remove expressions equal to 0. This means that it has to generate a much larger number of operations to compute a partial derivative. In this experiment, the MLIR code generated by the forward mode algorithm was 806 lines long, while the reverse mode code was only 94 lines long. After optimization, the forward mode code is 490 lines long, while the reverse mode code is 327 lines long. However, the execution times of both algorithms are almost the same.

## 6 Conclusion

In this work, forward mode and reverse mode automatic differentiation were implemented and evaluated in LingoDB using the MLIR compiler infrastructure. In our experiments, the reverse mode automatic differentiation algorithm outperformed the forward mode. Since MLIR evaluates the generated code at runtime, the system has to generate and handle more code when using the forward mode compared to the reverse mode, resulting in a performance discrepancy. However, as LingoDB is still in the early stages of development, these results are not yet applicable to real-world in-database machine learning use cases. It does not currently support the ability to call the implemented table functions, either iteratively or recursively. It also does not yet support matrices as a data type. Adding these features to LingoDB would create a promising system that allows for highly efficient machine learning capabilities directly within the database system.

```
select * from derivateForwards(TABLE(iris), TABLE(select random() as w1, random
      () as w2, random() as w3, random() as w4, random() as b), lambda{iris, x
      }(1 / (1 + exp(-(iris.sepal_length * x.w1 + iris.sepal_width * x.w2 +
      iris.petal_length * x.w3 + iris.petal_width * x.w4 + x.b)))));
```

**Listing 3: The performance test query. It is used twice: once as DerivateForwards and once as DerivateBackwards.**

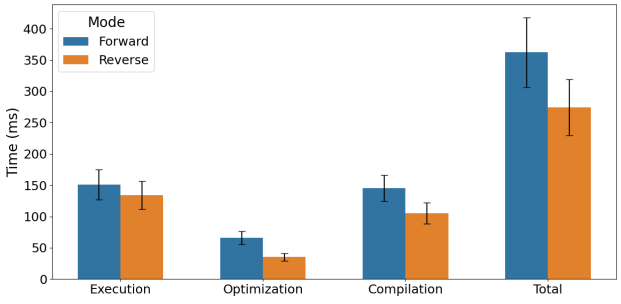



**Figure 9: The processing times of both forward (blue) and reverse (orange) mode calculating the derivatives of RSMLE, grouped by execution, optimization, compilation and total time.**

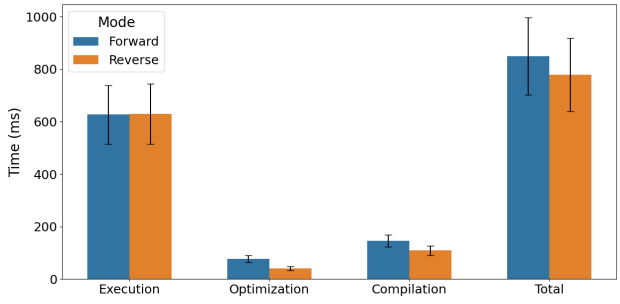



**Figure 10: The processing times of both forward (blue) and reverse (orange) mode using the performance test query, grouped by execution, optimization, compilation and total time.**

## References


[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.* 18, 1 (Jan. 2017), 5595–5637.

[3] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf

[4] Maximilian Böther, Ties Robroek, Viktor Gsteiger, Robin Holzinger, Xianzhe Ma, Pinar Tözün, and Ana Klimovic. 2025. Modyn: Data-Centric Machine Learning Pipeline Orchestration. *Proc. ACM Manag. Data* 3, 1 (2025), 55:1–55:30. https://doi.org/10.1145/3709705

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

[6] R. A. Fisher. 1936. The Use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics* 7, 2 (1936), 179–188. https://doi.org/10.1111/j.1469-1809.1936.tb02137.x

[7] The Apache Software Foundation. 2025. Apache Arrow. https://arrow.apache.org/ Accessed: February 11, 2025.

[8] Inc. Free Software Foundation. 2014. GNU Bison. https://www.gnu.org/software/bison/ Accessed: February 16, 2025.

[9] Yordan Grigorov, Haralampos Gavriilidis, Sergey Redyuk, Kaustubh Beedkar, and Volker Markl. 2023. P2D: A Transpiler Framework for Optimizing Data Science Pipelines. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning, DEEM 2023, Seattle, WA, USA, 18 June 2023*. ACM, 3:1–3:4. https://doi.org/10.1145/3595360.3595853

[10] Zezhou Huang, Rathijit Sen, Jiaxiang Liu, and Eugene Wu. 2023. JoinBoost: Grow Trees Over Normalized Data Using Only SQL. *Proc. VLDB Endow.* 16, 11 (2023), 3071–3084. https://doi.org/10.14778/3611479.3611509

[11] Michael Jungmair and Jana Giceva. 2023. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow.* 16, 11 (July 2023), 3461–3474. https://doi.org/10.14778/3611479.3611539

[12] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an open framework for query optimization and compilation. *Proc. VLDB Endow.* 15, 11 (July 2022), 2389–2401. https://doi.org/10.14778/3551793.3551801

[13] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[14] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[15] Yiming Lin and Sharad Mehrotra. 2024. PLAQUE: Automated Predicate Learning at Query Time. *Proc. ACM Manag. Data* 2, 1 (2024), 46:1–46:25. https://doi.org/10.1145/3639301

[16] Charles C. Margossian. 2019. A review of automatic differentiation and its efficient implementation. *WIREs Data Mining and Knowledge Discovery* 9, 4 (2019), e1305. https://doi.org/10.1002/widm.1305

[17] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[18] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath (Eds.). LNI, Vol. P-241. GI, 383–402. https://dl.gi.de/handle/20.500.12116/2418

[19] Sebastian Schelter, Stefan Grafberger, Shubha Guha, Bojan Karlas, and Ce Zhang. 2023. Proactively Screening Machine Learning Pipelines with ARGUSEYES. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*, Sudipto Das, Ippokratis Pandis, K. Selçuk Candan, and Sihem Amer-Yahia (Eds.). ACM, 91–94. https://doi.org/10.1145/3555041.3589682

[20] Maximilian E. Schüle and Jakob Hornung. 2024. Higher-Order SQL Lambda Functions. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 5622–5628. https://doi.org/10.1109/ICDE60146.2024.00450

[21] 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 2020: 32nd International Conference on Scientific and Statistical Database Management, Vienna, Austria, July 7-9, 2020*, Elaheh Pourabbas, Dimitris Sacharidis, Kurt Stockinger, and Thanasis Vergoulis (Eds.). ACM, 6:1–6:12. https://doi.org/10.1145/3400903.3400915

[22] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. 2021. In-Database Machine Learning with SQL on GPUs. In *SSDBM 2021: 33rd International Conference on Scientific and Statistical Database Management, Tampa, FL, USA, July 6-7, 2021*, Qiang Zhu, Xingquan Zhu, Yicheng Tu, Zichen Xu, and Anand Kumar (Eds.). ACM, 25–36. https://doi.org/10.1145/3468791.3468840

[23] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. 2022. Recursive SQL and GPU-support for in-database machine learning. *Distributed Parallel Databases* 40, 2-3 (2022), 205–259. https://doi.org/10.1007/S10619-022-07417-7

[24] Maximilian E. Schüle, Frédéric Simonis, Thomas Heyenbrock, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings (LNI, Vol. P-289)*, Torsten Grust, Felix Naumann, Alexander Böhm, Wolfgang Lehner, Theo Härder, Erhard Rahm, Andreas Heuer, Meike Klettke, and Holger Meyer (Eds.). Gesellschaft für Informatik, Bonn, 247–266. https://doi.org/10.18420/BTW2019-16

[25] Maximilian E. Schüle, Maximilian Springer, Alfons Kemper, and Thomas Neumann. 2022. LLVM code optimisation for automatic differentiation: when forward and reverse mode lead in the same direction. In *Proceedings of the Sixth Workshop on Data Management for End-To-End Machine Learning* (Philadelphia, Pennsylvania) *(DEEM '22)*. Association for Computing Machinery, New York, NY, USA, Article 5, 4 pages. https://doi.org/10.1145/3533028.3533302

[26] James Stanier and Des Watson. 2013. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.* 45, 3, Article 26 (July 2013), 27 pages. https://doi.org/10.1145/2480741.2480743

[27] Jacopo Tagliabue, Ryan Curtin, and Ciro Greco. 2024. FaaS and Furious: abstractions and differential caching for efficient data pre-processing. In *IEEE International Conference on Big Data, BigData 2024, Washington, DC, USA, December 15-18, 2024*, Wei Ding, Chang-Tien Lu, Fusheng Wang, Liping Di, Kesheng Wu, Jun Huan, Raghu Nambiar, Jundong Li, Filip Ilievski, Ricardo Baeza-Yates, and Xiaohua Hu (Eds.). IEEE, 3562–3567. https://doi.org/10.1109/BIGDATA62323.2024.10825377