Machine Learning, Linear Algebra, and More: Is SQL All You Need?

Mark Blacher, Joachim Giesen, Sören Laue, Julien Klaus
University of Jena
{mark.blacher,joachim.giesen,soeren.laue,julien.klaus}@uni-jena.de

Viktor Leis
University of Erlangen-Nuremberg
viktor.leis@fau.de

ABSTRACT

SQL is the standard language for retrieving and manipulating relational data. Although SQL is ubiquitous for simple analytical queries, it is rarely used for more complex computations like machine learning, linear algebra, and other computationally-intensive algorithms. These algorithms are often programmed in a procedural fashion and look very different from declarative SQL queries. However, SQL actually does provide constructs to perform all kinds of computations. In this paper, we show how to translate procedural constructs to SQL – enabling complex SQL-only algorithms. Using SQL for algorithms keeps computations close to the data, requires minimal user permissions, and increases software portability. The performance of the resulting SQL algorithms depends heavily on the underlying DBMS and the SQL code. Surprisingly, we find that query engines like HyPer can achieve very high performance – in some cases even outperforming state-of-the-art linear algebra packages like NumPy.

1 INTRODUCTION

It is well-known that SQL is Turing complete [1] and that, theoretically, arbitrary computations can be expressed in SQL. However, this is usually considered to be a theoretical observation rather than a practical approach. One hurdle to directly expressing complex algorithms in SQL is that algorithms are usually expressed in a procedural language. The declarative nature of SQL makes it non-trivial to write queries that perform complex, algorithmic computations like statistical learning or optimization algorithms.

Currently, complex algorithms are implemented outside of database systems, using user defined functions (UDFs), or by relying on system-specific in-DBMS operators. Expressing algorithms directly in SQL would have four major benefits:

(i) Near-data computation: Transferring data to external applications can be expensive. With SQL algorithms, the data remains in the database and the SQL query engine can start computations immediately. In addition, higher data privacy can be ensured if only the results of computations, but not the underlying data, are visible.

(ii) Flexibility: Many DBMSs provide in-house implementations of algorithms that allow users to perform advanced analytical computations. However, the spectrum of implemented algorithms varies greatly from DBMS to DBMS. Furthermore, customizability and extensibility of in-house implementations are limited. SQL algorithms, on the other hand, can be tailored specifically to use cases and modified as needed. Unlike stored procedures and user defined functions, SQL algorithms require only minimal user permissions. Users with only read permissions can perform all kinds of computations.

(iii) High degree of abstraction: SQL is a language that abstracts strongly from the architectural details of a DBMS. Vectorized, parallel or even distributed execution of SQL algorithms is done automatically by the underlying DBMS.

(iv) Portability: If SQL algorithms use a common SQL subset supported by multiple DBMS vendors, then the algorithms can be run on other DBMSs without modifications.

How to map algorithmic primitives such as variables, functions, conditions, loops, and error handling to SQL is not obvious. In this paper, we show that by using UNION ALL, the WITH clause, and its even more powerful extension WITH RECURSIVE, the basic imperative primitives for writing algorithms can be expressed in SQL. The performance of these SQL algorithms strongly depends on the DBMS and the structure of SQL code itself. Using a logistic regression case study, we present two styles for writing linear algebra code in SQL. It turns out that by choosing a database-friendly SQL coding style compiled queries can be even faster than procedural code written in NumPy. Thus, SQL-only analytics is not only possible, but practical.

The source code for the algorithmic primitives examples and the case study from the paper can be downloaded from https://github.com/mark-blacher/sql-algorithms. We chose logistic regression for our case study primarily for expositionary and space reasons. To show that our approach is applicable to far more complex algorithms, the repository includes an SQL-only Linear Programming solver. Other computationally-intensive algorithms, including Neural Networks, can be solved straightforwardly using our approach as well.

2 RELATED WORK

Much work has been done to incorporate compute-intensive analytics into database systems [2]. Kawaguchi et al. implement a Linear Programming solver using stored procedures [3]. Cohen et al. use a set of UDFs as library functions to realize complex analytical tasks such as solving a system of linear equations or computing ordinary least squares problems [4]. Raasveldt et al. integrate machine learning pipelines into MonetDB by using vectorized UDFs [5]. Schüle et al. incorporate gradient descent and tensor data types into HyPer to enable machine learning tasks such as regression, clustering, and classification [6]. However, these approaches rely on external tools, non-portable UDFs or require changes to the DBMS itself.

Du [7] and Hirn and Grust [8] present approaches for SQL-only analytics. Du shows how to implement common deep learning operations in SQL [7]. Du uses what we refer to in this paper as *COO style linear algebra* to implement the classification parts

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2022. 12th Annual Conference on Innovative Data Systems Research (CIDR '22). January 9-12, 2022, Chaminade, USA.

of a Convolutional Neural Network and a Graph Convolutional Network. The classification part of a neural network (the forward pass) is a sequence of different tensor operations and does not require iterative constructs. Still, his ideas can be used to run already trained neural networks in the database, using nothing but SQL.

Hirn and Grust compile PL/SQL UDFs to plain SQL queries [8]. On PostgreSQL the plain SQL queries improve execution times by about a factor of two compared to their PL/SQL UDFs counterparts. The speedup comes from avoiding PL/SQL inherent frictions in the computations such as context switches between SQL and PL/SQL parts of the code. Hirns and Grusts approach of compiling PL/SQL into SQL is very promising and shows us that a compiler could also be something for us to pursue in the future. Also, Ramachandra et al. [9] demonstrate similar techniques for compiling imperative UDFs into SQL. However, our goal is not to speed up UDFs. Our goal is to demonstrate how a procedurally-minded, SQL-savvy developer can efficiently translate their ideas directly into plain SQL algorithms. Furthermore, there are conceptual differences between Hirn and Grust, and us, in terms of the SQL style that is used to write algorithms. Hirn and Grust use, for example, LATERAL subqueries to implement challenging iterative constructs, whereas we use recursive references in subqueries instead. Overall, we think that it takes both a performance-aware SQL developer to write the bottleneck part of the algorithm and a compiler to create the boilerplate code around it.

3 SQL ALGORITHM PRIMITIVES

In this section, we map algorithmic primitives of procedural languages to SQL's declarative syntax. To demonstrate the translation, we show code snippets in Python and their counterparts in PostgreSQL's SQL dialect.

3.1 Variables

In SQL, variables can be represented as relations or as values within relations. Relations are suitable for modeling all required data structures for computations, such as scalars, vectors, matrices, tensors, sets, hash tables, and even trees and graphs [1, 7]. During a computation in SQL, new variables can be created using the WITH clause. The WITH clause allows one to name subqueries. These named subqueries can then be referenced in several places within the main query. However, unlike variables in procedural languages, variables created with the WITH clause in SQL are immutable. To update an SQL variable, a new variable must be created (see Listing 1).

Listing 1: Creating and updating variables.

```python
x = 1; x += 1; print(x) # x is mutable
```

```sql
WITH x(x) AS (       -- x is immutable
  VALUES (1)
), x2(x) AS (        -- create new variable
  SELECT x + 1 FROM x
) SELECT x FROM x2; -- print result
```

3.2 Functions

Functions are essential in most programming languages. They enable code reuse and parameter passing. While the SQL standard nowadays allows creating SQL functions, this is not supported by all systems and often requires special permissions. A practical alternative is to use the WITH construct, which allows embedding local functions into a SQL query (see Listing 2).

Listing 2: Calling functions.

```python
def kelvin_to_celsius(temperature):
  return [k - 273.15 for k in temperature]
def kelvin_to_fahrenheit(temperature):
  return [k * 9 / 5 - 459.67 for k in temperature]
temperature = [300, 170]
print(kelvin_to_fahrenheit(temperature))
```

```sql
WITH params(temperature) AS (
  VALUES (300::float), (170)
), kelvin_to_celsius(temperature) AS (     -- inline
  SELECT temperature - 273.15 FROM params
), kelvin_to_fahrenheit(temperature) AS ( -- inline
  SELECT temperature * 9 / 5 - 459.67 FROM params
) SELECT * FROM kelvin_to_fahrenheit;
```

Note that Listing 2 contains two functions for converting Kelvins. The user selects the desired conversion method. In this example, the user chooses to convert Kelvin to Fahrenheit. Programs written in SQL can therefore provide interfaces similar to procedural programs that allow users to choose the desired computations. From the developer's perspective, additional functions in SQL code can be used for debugging purposes.

3.3 Conditions

Standard SQL does not provide branching constructs, e.g. if-else, to control the program flow. In SQL, the construct that is closest to an if-else statement is the CASE statement. However, the CASE statement determines the results of expressions and therefore resembles a ternary operator rather than a control structure.

To emulate conditional control flow in SQL the UNION ALL construct is suitable. The UNION ALL construct combines the results of two or more SELECT statements. By combining only those results that satisfy conditions in the WHERE clauses of the SELECT statements, it is possible to emulate conditional control flow (see Listing 3). The only restriction for UNION ALL is that the number of columns and the data types of the columns in the individual SELECT statements must match.

Listing 3: Conditional assignment.

```python
if random.random() > 0.5: # random condition
  A = kelvin_to_celsius(temperatures)
else:
  A = kelvin_to_fahrenheit(temperatures)
print(A)
```

```sql
...
), condition AS ( -- random condition
  VALUES (random() > 0.5)
), A(A) AS (      -- only one relation is selected
  SELECT * FROM kelvin_to_celsius WHERE
    (SELECT * FROM condition)
  UNION ALL
  SELECT * FROM kelvin_to_fahrenheit WHERE
    NOT (SELECT * FROM condition)
) SELECT * FROM A
```

3.4 Loops

Expressing loops in complex computations by recursive queries is gradually becoming common practice [8, 10]. In SQL, there are two variants of loops that are realized by recursive queries and comply with the SQL standard [11]. The first variant is supported by most DBMSs and is a simple loop without recursive references in subqueries (see Listing 4). The second variant contains recursive references in subqueries and, to our knowledge, is fully supported only in PostgreSQL, DuckDB, and HyPer (see Listing 5).

Listing 4: Loop without recursive reference in a subquery.

```python
# Taylor series for approximating e^x
from math import factorial
x = [1.5, 4.0]
e_pow_x = [1.0] * len(x) # make list of ones
for i in range(1, 20):
  for j in range(len(x)):
    e_pow_x[j] += x[j] ** i / factorial(i)
print(e_pow_x) # e^x = 1 + x^1/1! + x^2/2! + ...
```

```sql
WITH RECURSIVE exp(x, i, e_pow_x) AS (
  VALUES (1.5::float, 1, 1.0::float),
         (4.0       , 1, 1.0       )
  UNION ALL
  SELECT x, i + 1, e_pow_x + POWER(x, i) / !!i
    FROM exp WHERE i < 20 -- no recursive reference
) SELECT e_pow_x FROM exp WHERE i = 20;
```

The second loop variant allows the recursive working table, x in Listing 5, to be used within subqueries in the FROM clause. These subqueries may also be recursive. PostgreSQL generates an error if the recursive reference to the working table appears more than once within the FROM clause. This error can be avoided by creating a new variable within the FROM clause and referencing it in further computations (see *workaround for PostgreSQL* in Listing 5). This workaround allows referencing the recursive working table multiple times.[1] We heavily exploit recursive references in subqueries to implement all kinds of algorithms in SQL. By supporting recursive references in subqueries, DBMS vendors could enable SQL-only algorithms in their products.

The limiting factor in recursive queries is the lacking possibility of having multiple working tables. Only one working table is allowed within the recursion. If an algorithm needs to update multiple variables in each iteration, they must all be packed into the working table at once. During an iteration, these variables then must be unpacked from the working table and afterwards packed again for the next iteration.

3.5 Errors

The input data for an algorithm may be incorrect. SQL programs of practical value should provide feedback on incorrect input data. To implement input validation in SQL, we use the UNION ALL construct. In Listing 6, the entropy of a probability distribution on three states is computed. A property of probabilities is that they are greater than or equal to zero and their sum over all states is one. These preconditions are checked in the WHERE clauses when creating the `errors` relation. If an erroneous input is detected, the

[1]This workaround is the missing link to complete Reprintsev's Turing completeness proof for SQL [12].

Listing 5: Loop with recursive references in subqueries.

```python
# Newton's method: compute x so that
# f(x) = x^2 + cos(x) + 2 * sin(x) is minimized
from math import sin, cos
x = 0
while True:
  fD = 2 * x - sin(x) + 2 * cos(x) # f'
  if abs(fD) < 1e-6:
    break
  fD2 = 2 - cos(x) - 2 * sin(x)    # f''
  x -= fD / fD2                    # update x
print(x)                           # -0.975012
```

```sql
WITH RECURSIVE x(x, i) AS (
  VALUES (0::float, 0::int)
UNION ALL
  SELECT x, i + 1 FROM (
    WITH x(x, i) AS ( -- recursive reference of x
      SELECT * FROM x -- workaround for PostgreSQL
    ), fD(fD) AS (    -- f'
      SELECT 2 * x - SIN(x) + 2 * COS(x) FROM x
    ), fD2(fD2) AS (  -- f''
      SELECT 2 - COS(x) - 2 * SIN(x) FROM x
    ) SELECT x - fD / fD2, fD, i FROM x, fD, fD2
  ) AS x_new(x, fD, i) WHERE ABS(fD) > 1e-6
) SELECT x FROM x WHERE i = (SELECT MAX(i) FROM x);
```

`errors` relation contains the corresponding error messages. In our example, the entropy is only computed if the `errors` relation is empty.

Listing 6: Error handling in the input data.

```python
from math import log2
probs = [0.1, 0.4, 0.5]
errors = []
if any(p < 0 for p in probs) == True:
  errors += ["ERROR: negative probabilities"]
if abs(sum(probs) - 1.0) > 1e-16:
  errors += ["ERROR: sum of probabilities != 1.0"]
if len(errors):
  print(*errors, sep='\n')
else:
  entropy = -sum([p * log2(p) for p in probs])
  print(entropy)
```

```sql
WITH probs(p) AS (
  VALUES (0.1::float), (0.4), (0.5)
), errors AS ( -- table contains all errors
  SELECT 'ERROR: negative probabilities'
    WHERE (SELECT 0 > ANY(SELECT * FROM probs))
  UNION ALL
  SELECT 'ERROR: sum of probabilities != 1.0'
    WHERE (SELECT ABS(SUM(p) - 1.0) > 1e-16
      FROM probs)
), entropy(entropy) AS ( -- compute if no errors
  SELECT -SUM(p * LOG(p) / LOG(2.0)) FROM probs
    WHERE NOT EXISTS(SELECT 1 FROM errors)
), result(result) AS (
  SELECT * FROM errors
  UNION ALL
  SELECT entropy::text FROM entropy
) SELECT * FROM result WHERE result IS NOT NULL
```

4 CASE STUDY

We use gradient descent-based logistic regression as a case study to demonstrate that SQL algorithms written in a database-friendly style can be practical. Logistic regression is a popular machine learning method for binary classification that leads to the following convex optimization problem:

$$\min_{w} 0.5 \cdot w^{\top} \cdot w + c \cdot \text{sum}(\log(\mathbf{1} + \exp(-y \odot (X \cdot w)))).$$

Here $c \in \mathbb{R}$ is a regularization parameter, $X \in \mathbb{R}^{n \times m}$ a data matrix with n data points (examples) and m features, whereby each data point i is assigned a label $y_i \in \{-1, 1\}$. The operator $\odot$ denotes element-wise multiplication.

To find the solution vector $w \in \mathbb{R}^m$ that minimizes the given objective function, gradient descent can be used as a numerical optimization algorithm. Minimization in gradient descent is achieved by repeatedly updating the parameter vector w using the following formula:

$$w(t+1) = w(t) - \alpha \frac{\partial f(w)}{\partial w},$$

where $\frac{\partial f(w)}{\partial w}$ is the derivative of the objective function with respect to the parameter vector w, that is, the gradient, and α the learning rate. To compute the derivative of the objective function, we use *Matrix Calculus*, an online tool for computing vector and matrix derivatives [13–15]. The full gradient descent-based logistic regression algorithm that we use for the case study is shown in Listing 7.

Listing 7: Gradient descent-based logistic regression.

```python
def gradient(X, c, w, y):
  cse = np.exp(-(y * X.dot(w)))
  return w - c * X.T.dot(cse * y / (1 + cse))

def gradient_descent(X, y, c, iterations):
  w = np.zeros(X.shape[1])  # initial weights
  alpha = 0.001             # learning rate
  for i in range(iterations):
    w = w - alpha * gradient(X, c, w, y)
  return w

w = gradient_descent(X, y, c=2, iterations=100)
```

4.1 Database-friendly SQL mappings

Computing the gradient is the most time-demanding operation of the algorithm in Listing 7. If performance is important, the SQL code for computing the gradient must be database friendly. Here, the gradient computation consists for the most part of linear algebra operations. Linear algebra computations in SQL are usually mapped to a format that explicitly stores the indices for each value of a vector or matrix. This representation of vectors and matrices is similar to the coordinate format (COO) used for sparse linear algebra. Listing 8 shows how to compute the gradient from Listing 7 using the COO style in SQL.

An advantage of using the COO style for linear algebra in SQL is its universality, that is, there is no dependence of the SQL code on the number of columns in a matrix. Sparse linear algebra is also supported by default when using COO style. Furthermore, active research is being conducted to develop query engines that reduce the execution times of COO-like linear algebra queries [16]. However, performance of COO-style SQL code may be insufficient because of increased memory consumption due to the explicit indices, poor locality, and costly transformations that get data into the right format. Moreover, COO-style linear algebra in SQL relies heavily on joins. In Listing 8 joins are specified in the WHERE clauses.

Listing 8: COO style gradient computation.

```
WITH X(i, j, val) AS ( -- dataset and ones
  VALUES (0::int, 0::int, 5.5::float),
                        (0, 1, 2.4),  (0, 2, 1.0),
         (1, 0, 5.4),   (1, 1, 3.0),  (1, 2, 1.0),
         (2, 0, 5.5),   (2, 1, 4.2),  (2, 2, 1.0)
), y (i, val) AS (      -- labels
  VALUES (0::int, 1.0::float), (1, 1.0), (2, -1.0)
), w (i, val) AS (      -- weights
  VALUES (0::int, 0.1::float), (1, 0.1), (2, 0.1)
), Xw(i, val) AS (      -- X.dot(w)
  SELECT X.i, SUM(X.val * w.val) FROM X, w
    WHERE X.j = w.i GROUP BY X.i
), cse(i, val) AS (     -- np.exp(-(y * Xw))
  SELECT y.i, EXP(-(y.val * Xw.val)) FROM y, Xw
    WHERE y.i = Xw.i
), v(i, val) AS (       -- cse * y / (1 + cse)
  SELECT y.i, cse.val * y.val / (1 + cse.val)
    FROM cse, y WHERE cse.i = y.i
), u(i, val) AS (       -- c * X.T.dot(v), c = 2
  SELECT X.j, 2 * SUM(X.val * v.val) FROM X, v
    WHERE X.i = v.i GROUP BY X.j
), g(i, val) AS (       -- w - u
  SELECT w.i, w.val - u.val FROM w, u
    WHERE w.i = u.i
) SELECT * FROM g       -- gradient
```

An alternative approach to COO-style linear algebra in SQL is to treat relations themselves as vectors or matrices. Rows and columns of the relation correspond to row and column vectors of linear algebra. This representation of vectors and matrices is similar to how they are represented in dense linear algebra. Listing 9 shows the gradient computation from Listing 7 using the dense linear algebra style in SQL.

Listing 9: Database-friendly gradient computation.

```
WITH X(f1, f2, y) AS (    -- dataset and labels
  VALUES (5.5::float, 2.4::float, 1.0::float),
         (5.4,        3.0,        1.0       ),
         (5.5,        4.2,       -1.0       )
), w(w1, w2, intercept) AS ( -- weights
  VALUES (0.1::float, 0.1::float, 0.1::float)
), cse(val, f1, f2) AS ( -- np.exp(-(y * X.dot(w)))
  SELECT exp((f1 * w1 + f2 * w2 + intercept) * -y),
    f1, f2, y FROM X, w   -- propagate f1, f2, y
), v(val, f1, f2) AS (    -- cse * y / (1 + cse)
  SELECT val * y / (1.0 + val), f1, f2 FROM cse
), u(t1, t2, t3) AS (     -- c * X.T.dot(v), c = 2
  SELECT 2 * SUM(f1 * val), 2 * SUM(f2 * val),
    2 * SUM(val) FROM v   -- use f1, f2
), g (g1, g2, g3) AS (    -- w - u
  SELECT w1 - t1, w2 - t2, intercept - t3 FROM w, u
) SELECT * FROM  g        -- gradient
```

The gradient computation in Listing 9 avoids joins and does not require explicit indices for vectors and matrices. We therefore call this computation database friendly. The features `f1`, `f2`, and the

labels `y` are stored in a single relation `X`. During the computation of the gradient, features and labels are propagated to avoid unnecessary joins. See, for example, how the features `f1`, `f2` are selected when the relations `cse` and `v` are created. These "unnecessary" selections avoid joins when `f1` and `f2` are needed again, as is the case, when creating the relation `u`.

4.2 Performance results

We compare the performance for the gradient descent-based logistic regression between NumPy, HyPer, and PostgreSQL. NumPy is a Python package for high performance scientific computing. We link NumPy against the Math Kernel Library (MKL) version 2020.0.2. Intel's MKL library makes extensive use of vector instructions and multicore processing. HyPer is a column-oriented in-memory DBMS that achieves high performance for both OLTP and OLAP workloads [17]. We use Tableau's publically available *Hyper API* version 0.0.13287. PostgreSQL is a widely used, open source, row-oriented DBMS. We use PostgreSQL version 12.8.

For our measurements, we use a machine with an Intel i9-10980XE 18-core processor (36 hyperthreads) running Ubuntu 20.04.1 LTS with 128 GB of RAM. Each core has a base frequency of 3.0 GHz and a max turbo frequency of 4.6 GHz, and supports the AVX-512 vector instruction set. For HyPer and PostgreSQL, we measure the performance of two different implementations of logistic regression. One is based on COO-style linear algebra, the other, the database-friendly one, uses the join-free dense-style for linear algebra (see previous subsection for details). For the database measurements, we use temporary tables. We do not use database indexes. We report performance in terms of iterations per second when running the logistic regression solver. One iteration computes the gradient and updates the weights of the previous iteration. We verify that all implementations compute identical weights and thus achieve identical model accuracy.

Figure 1 shows the performance of gradient descent-based logistic regression for a dataset with 32 features and 1 000 000 examples. While the database-friendly implementation on HyPer achieves over 100 iterations per second, the performance on PostgreSQL is no different from its COO implementation. The performance on PostgreSQL is very low compared to HyPer because HyPer is a highly efficient parallel DBMS that uses query compilation. We therefore ignore PostgreSQL in further measurements. Surprisingly, however, HyPer is almost three times faster than NumPy. The following measurements explore the conditions under which HyPer outperforms NumPy.

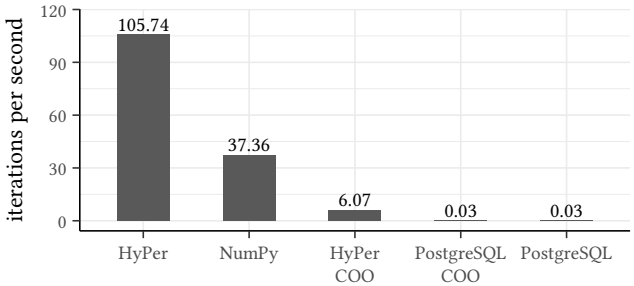

Figure 1: Performance for HyPer, PostgreSQL, and NumPy (32 features, 1 000 000 examples, multi-threaded).

Figure 2 shows the performance depending on the number of threads used for the computation. When using two threads, HyPer and NumPy perform about the same. When using more than two threads, the database-friendly implementation on HyPer is faster than NumPy. HyPer steadily gains performance as the number of available cores increases (threads $\leq$ 18). HyPer gains even more performance by additional parallelization through hyperthreads. In contrast to HyPer, the parallelization of a gradient computation in NumPy is poor. Although the MKL is generally very efficient at parallelizing linear algebra operations, it fails for the 32-feature example in Figure 2.

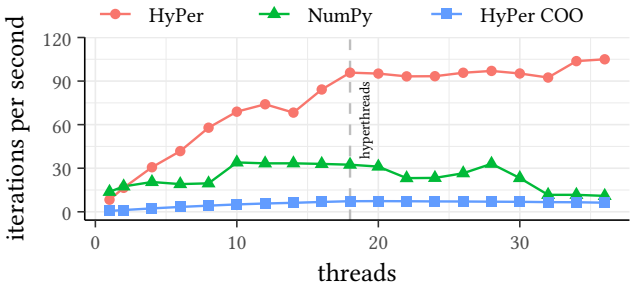

Figure 2: Multi-threaded performance scalability for HyPer and NumPy (32 features, 1 000 000 examples).

In Figure 3, we vary the number of features between 4 and 128. We leave it up to the implementations to decide how many threads to use. NumPy utilizes all 18 cores with 4 features in the dataset, but as Figure 3 indicates, this happens rather inefficiently. Figure 3 shows that HyPer is the fastest alternative for fewer than 128 features in the dataset. However, the performance gap between HyPer and NumPy closes as the number of features in the dataset increases. At 128 features NumPy is even a bit faster than HyPer.

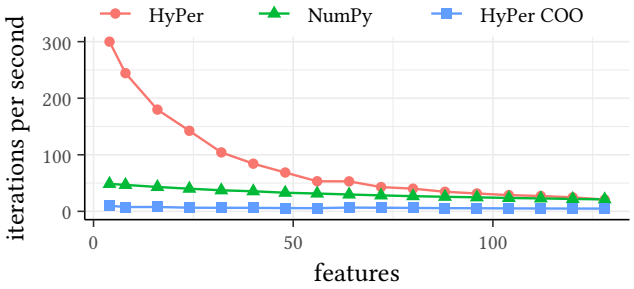

Figure 3: Performance based on the number of features for HyPer and NumPy (1 000 000 examples, multi-threaded).

Figure 4 shows the performance as a function of the number of examples in the dataset. The measurements are shown on a logarithmic scale. For less than or equal to 10^5 examples in the dataset, NumPy is the fastest option. NumPy stores matrices and vectors contiguously in memory and therefore benefits from good locality when the dataset fits in the cache. Furthermore, since a weights update depends on the previously computed weights, efficient parallelization does not come into play for small data sets. Also, for benchmarking, we ran the algorithm repeatedly for 100 iterations. In HyPer, these times include the runtimes for the cached SQL queries. The setup costs for a query do not amortize for small

amounts of data, therefore NumPy is significantly faster for small problem sizes. In HyPer, there is an irregularity in the measurements. HyPer executes more iterations per second for 10^6 examples than for 10^5 examples in the dataset. The reason for this irregularity is that HyPer executes all queries with less than or equal to 10^5 examples by only one thread.

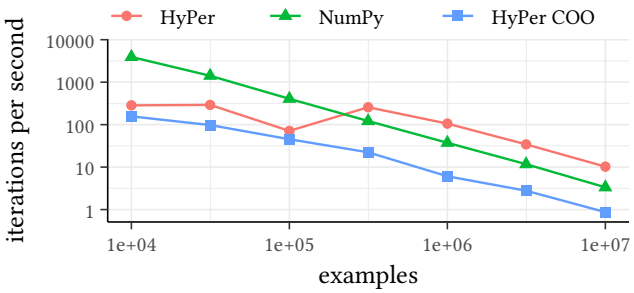

Figure 4: Performance based on the number of examples for HyPer and NumPy (32 features, multi-threaded).

4.3 How can HyPer be faster than NumPy?

HyPer's dynamic tuple-wise parallelization scheme can be very efficient for compute intensive analytics. Also, when compiling SQL queries in HyPer, the code is optimized as a whole. Parts that can be executed together, such as $\exp(-y \cdot (X \cdot w))$ in the gradient computation, are fused, that is pipelined, and not executed individually as in NumPy. This saves memory bandwidth and increases the cache locality of the computation. Our measurements suggest that HyPer can be faster than NumPy for large data sets with a limited number of columns. The use case with a large number of examples (rows) and a small number of features (columns) is common in databases. In these cases, SQL algorithms on modern DBMSs may even have similar or better performance than hand-optimized procedural alternatives.

To achieve maximum performance, SQL algorithms must be written in a way that exploits the architecture of the underlying DBMS as efficiently as possible. In the database-friendly logistic regression variant, the data-intensive computations of the gradient are performed on the columns (see Listing 9). Therefore, on modern column-oriented DBMSs like HyPer, this SQL coding style is efficient due to its good data locality, vectorization and parallelization capabilities. But, a word of caution needs to be said about the dense linear algebra style in SQL. We avoided joins in the computation. If joins cannot be avoided, additional row indices must be introduced, which can render the computation less performant.

5 CONCLUSIONS AND FUTURE WORK

SQL-only algorithms are not a theoretical gimmick, but can be of high practical value. They offer benefits such as near-data computations, flexibility for code changes, a high degree of abstraction from the underlying DBMS architecture, and portability. We showed how algorithmic primitives can be expressed in SQL. By using these primitives, computationally-intensive algorithms can be implemented in SQL. In the case study, we presented database-friendly SQL code that avoids joins for linear algebra operations. It turned out that the database-friendly SQL implementation on HyPer can outperform even NumPy on larger data sets.

Loops in algorithms often contain sophisticated computations with data from previous iterations. We demonstrated how to implement such loops in SQL by using recursive references to the working table in the FROM clause of WITH RECURSIVE. DBMSs like PostgreSQL, DuckDB or HyPer that support recursive references, enable already all kinds of computations with SQL. Extending a DBMS to support recursive references in subqueries would be straightforward, and we hope that this paper triggers DBMS developers to implement this feature.

In future work, we plan to automate the translation of imperative constructs to SQL using compilation. A compiler approach is also conceivable to translate linear algebra operations to SQL. In order to make the translation of linear algebra operations as efficient as possible, the limits of join-free linear algebra in SQL need to be further explored. Deep Learning also seems to be an interesting use case that we plan to explore. By computing derivatives with external tools like our *Matrix Calculus* [13–15] and translating them to SQL, the only thing left to do is implementing an optimization algorithm in SQL, and this is possible, as we have shown in the case study. Therefore, we believe that SQL machine learning and other forms of compute intense analytics in SQL are highly promising.

ACKNOWLEDGMENTS

This work was supported by the German Science Foundation (DFG) grant (GI-711/5-1) within the priority program (SPP1736) *Algorithms for Big Data*, and by the Carl Zeiss Foundation within the project *A Virtual Werkstatt for Digitization in the Sciences*.

REFERENCES

[1] D. Fetter, "Lists and recursion and trees, oh my!." FOSDEM: Free and Open Source Software Developers' European Meeting, 2010.
[2] M. Boehm, A. Kumar, and J. Yang, *Data Management in Machine Learning Systems*. 2019.
[3] A. Kawaguchi and J. A. Perez, "Linear programming for database environment," in *ICINCO*, 2007.
[4] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton, "MAD skills: New analysis practices for big data," *Proc. VLDB Endow.*, 2009.
[5] M. Raasveldt, P. Holanda, H. Mühleisen, and S. Manegold, "Deep integration of machine learning into column stores," in *EDBT*, 2018.
[6] M. E. Schüle, F. Simonis, T. Heyenbrock, A. Kemper, S. Günnemann, and T. Neumann, "In-database machine learning: Gradient descent and tensor algebra for main memory database systems," in *BTW*, 2019.
[7] L. Du, "In-machine-learning database: Reimagining deep learning with old-school SQL," *arXiv:2004.05366*, 2020.
[8] D. Hirn and T. Grust, "One WITH RECURSIVE is worth many GOTOs," in *SIGMOD*, 2021.
[9] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. A. Galindo-Legaria, and C. Cunningham, "Froid: Optimization of imperative programs in a relational database," *Proc. VLDB Endow.*, 2017.
[10] M. E. Schüle, H. Lang, M. Springer, A. Kemper, T. Neumann, and S. Günnemann, "In-database machine learning with SQL on gpus," in *SSDBM*, 2021.
[11] ISO/IEC 9075-2:2016, *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*. 2016.
[12] A. Reprintsev, "Turing completeness," in *Oracle SQL Revealed*, 2018.
[13] S. Laue, M. Mitterreiter, and J. Giesen, "Computing higher order derivatives of matrix and tensor expressions," in *NeurIPS*, 2018.
[14] S. Laue, M. Mitterreiter, and J. Giesen, "GENO - generic optimization for classical machine learning," in *NeurIPS*, 2019.
[15] S. Laue, M. Mitterreiter, and J. Giesen, "A simple and efficient tensor calculus," in *AAAI*, 2020.
[16] C. R. Aberger, A. Lamb, K. Olukotun, and C. Ré, "LevelHeaded: A unified engine for business intelligence and linear algebra querying," in *ICDE*, 2018.
[17] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *ICDE*, 2011.